

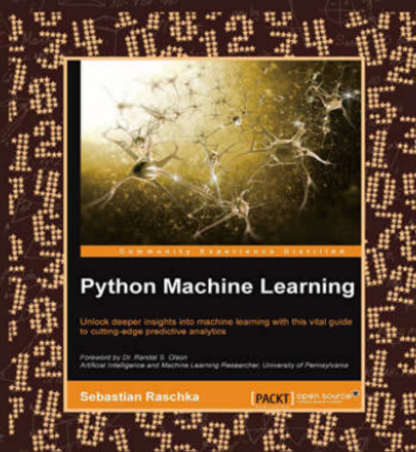
[PACKT] PUBLISHING

数据科学与工程技术丛书

Python机器学习

[美] 塞巴斯蒂安·拉施卡（Sebastian Raschka） 著

高明 徐莹 陶虎成 译



PYTHON MACHINE LEARNING



机械工业出版社
China Machine Press

数据科学与工程技术丛书

Python机器学习

Python Machine Learning

（美）塞巴斯蒂安·拉施卡（Sebastian Raschka） 著

高明 徐莹 陶虎成 译

ISBN：978-7-111-55880-4

本书纸版由机械工业出版社于2017年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

译者序

推荐序

作者简介

审校者简介

前言

第1章　赋予计算机学习数据的能力

1.1　构建智能机器将数据转化为知识

1.2　机器学习的三种不同方法

1.2.1　通过监督学习对未来事件进行预测

1.2.2　通过强化学习解决交互式问题

1.2.3　通过无监督学习发现数据本身潜在的结构

1.2.4　基本术语及符号介绍

1.3　构建机器学习系统的蓝图

1.3.1　数据预处理

1.3.2　选择预测模型类型并进行训练

1.3.3　模型验证与使用未知数据进行预测

1.4　Python在机器学习中的应用

本章小结

第2章　机器学习分类算法

2.1　人造神经元—早期机器学习概览

2.2　使用Python实现感知器学习算法

2.3　自适应线性神经元及其学习的收敛性

2.3.1　通过梯度下降最小化代价函数
2.3.2　使用Python实现自适应线性神经元
2.3.3　大规模机器学习与随机梯度下降
本章小结
第3章　使用scikit-learn实现机器学习分类算法
3.1　分类算法的选择
3.2　初涉scikit-learn的使用
3.3　逻辑斯谛回归中的类别概率
3.3.1　初识逻辑斯谛回归与条件概率
3.3.2　通过逻辑斯谛回归模型的代价函数获得权重
3.3.3　使用scikit-learn训练逻辑斯谛回归模型
3.3.4　通过正则化解决过拟合问题
3.4　使用支持向量机最大化分类间隔
3.4.1　对分类间隔最大化的直观认识
3.4.2　使用松弛变量解决非线性可分问题
3.4.3　使用scikit-learn实现SVM
3.5　使用核SVM解决非线性问题
3.6　决策树
3.6.1　最大化信息增益—获知尽可能准确的结果
3.6.2　构建决策树
3.6.3　通过随机森林将弱分类器集成为强分类器
3.7　惰性学习算法—k-近邻算法
本章小结
第4章　数据预处理——构建好的训练数据集

4.1　缺失数据的处理
4.1.1　将存在缺失值的特征或样本删除
4.1.2　缺失数据填充
4.1.3　理解scikit-learn预估器的API
4.2　处理类别数据
4.2.1　有序特征的映射
4.2.2　类标的编码
4.2.3　标称特征上的独热编码
4.3　将数据集划分为训练数据集和测试数据集
4.4　将特征的值缩放到相同的区间
4.5　选择有意义的特征
4.5.1　使用L1正则化满足数据稀疏化
4.5.2　序列特征选择算法
4.6　通过随机森林判定特征的重要性
本章小结
第5章　通过降维压缩数据
5.1　无监督数据降维技术—主成分分析
5.1.1　总体方差与贡献方差
5.1.2　特征转换
5.1.3　使用scikit-learn进行主成分分析
5.2　通过线性判别分析压缩无监督数据
5.2.1　计算散布矩阵
5.2.2　在新特征子空间上选取线性判别算法
5.2.3　将样本映射到新的特征空间

5.2.4　使用scikit-learn进行LDA分析
5.3　使用核主成分分析进行非线性映射
5.3.1　核函数与核技巧
5.3.2　使用Python实现核主成分分析
5.3.3　映射新的数据点
5.3.4　scikit-learn中的核主成分分析
本章小结
第6章　模型评估与参数调优实战
6.1　基于流水线的工作流
6.1.1　加载威斯康星乳腺癌数据集
6.1.2　在流水线中集成数据转换及评估操作
6.2　使用k折交叉验证评估模型性能
6.2.1　holdout方法
6.2.2　k折交叉验证
6.3　通过学习及验证曲线来调试算法
6.3.1　使用学习曲线判定偏差和方差问题
6.3.2　通过验证曲线来判定过拟合与欠拟合
6.4　使用网格搜索调优机器学习模型
6.4.1　使用网络搜索调优超参
6.4.2　通过嵌套交叉验证选择算法
6.5　了解不同的性能评价指标
6.5.1　读取混淆矩阵
6.5.2　优化分类模型的准确率和召回率
6.5.3　绘制ROC曲线

6.5.4　多类别分类的评价标准
本章小结
第7章　集成学习——组合不同的模型
7.1　集成学习
7.2　实现一个简单的多数投票分类器
7.3　评估与调优集成分类器
7.4　bagging—通过bootstrap样本构建集成分类器
7.5　通过自适应boosting提高弱学习机的性能
本章小结
第8章　使用机器学习进行情感分析
8.1　获取IMDb电影评论数据集
8.2　词袋模型简介
8.2.1　将单词转换为特征向量
8.2.2　通过词频-逆文档频率计算单词关联度
8.2.3　清洗文本数据
8.2.4　标记文档
8.3　训练用于文档分类的逻辑斯谛回归模型
8.4　使用大数据—在线算法与外存学习
本章小结
第9章　在Web应用中嵌入机器学习模型
9.1　序列化通过scikit-learn拟合的模型
9.2　使用SQLite数据库存储数据
9.3　使用Flask开发Web应用
9.3.1　第一个Flask　Web应用

9.3.2 表单验证及渲染
9.4 将电影分类器嵌入Web应用
9.5 在公共服务器上部署Web应用
本章小结
第10章 使用回归分析预测连续型目标变量
10.1 简单线性回归模型初探
10.2 波士顿房屋数据集
10.3 基于最小二乘法构建线性回归模型
10.3.1 通过梯度下降计算回归参数
10.3.2 使用scikit-learn估计回归模型的系数
10.4 使用RANSAC拟合高鲁棒性回归模型
10.5 线性回归模型性能的评估
10.6 回归中的正则化方法
10.7 线性回归模型的曲线化-多项式回归
10.7.1 房屋数据集中的非线性关系建模
10.7.2 使用随机森林处理非线性关系
本章小结
第11章 聚类分析——处理无类标数据
11.1 使用k-means算法对相似对象进行分组
11.1.1 k-means＋＋
11.1.2 硬聚类与软聚类
11.1.3 使用肘方法确定簇的最佳数量
11.1.4 通过轮廓图定量分析聚类质量
11.2 层次聚类

11.2.1　基于距离矩阵进行层次聚类
11.2.2　树状图与热度图的关联
11.2.3　通过scikit-learn进行凝聚聚类
11.3　使用DBSCAN划分高密度区域
本章小结
第12章　使用人工神经网络识别图像
12.1　使用人工神经网络对复杂函数建模
12.1.1　单层神经网络回顾
12.1.2　多层神经网络架构简介
12.1.3　通过正向传播构造神经网络
12.2　手写数字的识别
12.2.1　获取MNIST数据集
12.2.2　实现一个多层感知器
12.3　人工神经网络的训练
12.3.1　计算逻辑斯谛代价函数
12.3.2　通过反向传播训练神经网络
12.4　建立对反向传播的直观认识
12.5　通过梯度检验调试神经网络
12.6　神经网络的收敛性
12.7　其他神经网络架构
12.7.1　卷积神经网络
12.7.2　循环神经网络
12.8　关于神经网络的实现
本章小结

第13章　使用Theano并行训练神经网络
13.1　使用Theano构建、编译并运行表达式
13.1.1　什么是Theano
13.1.2　初探Theano
13.1.3　配置Theano
13.1.4　使用数组结构
13.1.5　整理思路—线性回归示例
13.2　为前馈神经网络选择激励函数
13.2.1　逻辑斯谛函数概述
13.2.2　通过softmax函数评估多类别分类任务中的类别概率
13.2.3　通过双曲正切函数增大输出范围
13.3　使用Keras提高训练神经网络的效率
本章小结

译者序

机器学习是一门研究如何使用计算机模拟人类行为，以获取新的知识与技能的学科。它是人工智能的核心，同时也是处理大数据的关键技术之一。机器学习的主要目标是自动地从数据中发现价值的模式，亦即将原始信息自动转换为人们可以加以利用的知识。

随着科技的进步，特别是互联网技术的发展，使得我们在不知不觉中被卷入了大数据时代。传统的方法已经无法处理如此庞大的数据量，而机器学习技术正是解决此问题的良方。

不同于晦涩的学术书籍，本书是为程序员而作，因此没有过多枯燥的理论讲解，而是借助于Python语言及其机器学习库scikit-learn来帮助程序员快速理解算法的核心与本质，并能在生产环境轻松地加以应用。

本书对机器学习的各种算法进行了系统的讲解。第1章对机器学习以及Python在机器学习中的应用进行了简要的介绍，在后续章节分别讨论了数据分类（第2、3章）、数据预处理（第4、5章）、模型优化（第6章）、集成学习（第7章）、回归（第10章）、聚类（第11章），以及当前流行的神经网络及其深度学习（第12、13章）。每个章节基本上都是按照算法介绍、Python实现，以及使用scikit-learn来应用算法这样的模式进行讨论的，让读者既能掌握算法的本质，又能尽快将其应用到实际开发中去。

第8、9章介绍了机器学习与其他技术相结合的使用情况。第8章通过

与自然语言处理方法相结合，以IMDb电影评论数据集作为信息来源，借助于文本处理技术，对用户的情感倾向进行了分析。第9章包含许多实用的主题，包括如何序列化训练得到的模型、使用SQLite存储数据等，并通过实例演示了如何通过Web来分享分类模型的使用。

翻译的过程本身也是一个学习提高的过程，我们已经尽量去保证译文的准确性，但错误仍旧在所难免，如有问题还恳请读者不吝指教。此外，在本书出版后，我们将与同济大学和深圳大学合作，分别在上海、深圳两地就本书的内容与读者进行多次交流与讲解活动，有兴趣的读者可以邮件联系：gaomingsz@vip.163.com。

在翻译本书的过程中得到了深圳市意行科技开发公司、上海市公安高等专科学校、深圳市六度人和科技有限公司等单位领导的支持，在此一并表示感谢。

高明

gaomingsz@vip.163.com

推荐序

我们生活在数据洪流中。根据最新的估计，每天会产生大约2.5艾（1018）字节的数据。这个数据量非常之大，其中超过90%都是近十年产生的。然而，大部分信息都没有得到充分的利用，究其原因，可能是因为数据量超出了常用分析方法的处理范围，也可能是因为人类智力有限，无法处理如此庞大的数据。

通过机器学习，我们可以使用计算机处理数据、对数据进行学习，从而打破大数据看似牢不可破的禁锢，并形成我们自己的见解。无论是支持谷歌搜索引擎的超级计算机，还是人们每天使用的智能手机，机器学习在日常生活中确实起到了不可或缺的作用，通常，我们并没意识到它的存在。

大数据的开拓者为人们打开了一个崭新的领域，我们有必要进一步了解机器学习。什么是机器学习？机器学习是如何工作的？如何利用机器学习去探索未知，处理个人事务，抑或是在互联网上找到自己最喜欢的电影？Sebastian Raschka是我的好友兼同事，本书不仅涵盖了上述几点，还包含更多的精彩内容。

Sebastian坚持不懈地将他的业余时间都贡献给了机器学习开源社区。过去几年，Sebastian已经完成了许多本使用Python进行机器学习及数据可视化的入门教程。他还是多个Python开源包的贡献者，其中有几个已经成为Python机器学习工作流中的核心组件。

Sebastian精通机器学习，我确信他对Python机器学习的深刻见解将会让不同层次的从业者从中收益。对于任何想开阔自己在机器学习领域的视野，或者想获得更多实操经验的读者来说，我强烈推荐此书。

Randal S.Olson博士

宾夕法尼亚大学人工智能与机器学习领域的研究员

作者简介

Sebastian Raschka是密歇根州立大学的博士生，他在计算生物学领域提出了几种新的计算方法，还被科技博客Analytics Vidhya评为GitHub上最具影响力的数据科学家。他有一整年都使用Python进行编程的经验，同时还多次参加数据科学应用与机器学习领域的研讨会。正是因为Sebastian在数据科学、机器学习以及Python等领域拥有丰富的演讲和写作经验，他才有动力完成此书的撰写，目的是帮助那些不具备机器学习背景的人设计出由数据驱动的解决方案。

他还积极参与到开源项目中，由他开发完成的计算方法已经被成功应用到了机器学习竞赛（如Kaggle等）中。在业余时间，他沉醉于构建体育运动的预测模型，要么待在电脑前，要么在运动。

首先，我要感谢Arun Ross和Pang-Ning Tan教授，以及那些曾经启发我并激起我在模式分类、机器学习、数据挖掘领域兴趣的人。

我还想借此机会对Python社区和开源包的开发者表示感谢，他们帮助我创建了一个用于科学研究和数据科学的完美开发环境。

在此，还要特别感谢scikit-learn的核心开发人员。作为此项目的一个参与者，我有幸与这些极客合作，他们不仅对机器学习有着深入的了解，同时还都是非常出色的程序员。

最后，我还要感谢所有对本书感兴趣的读者，也真心希望我的热情能够感染大家一起加入到Python与机器学习社区中来。

审校者简介

Richard Dutton 8岁就开始使用ZX Spectrum编程，对于编程的痴迷使得他深入探索了技术在金融领域中的应用。

他曾在微软工作，现在是Barclays的董事，正痴迷于Python、机器学习和区块链的综合应用。

Dave Julian是一位IT顾问，同时还是一位具有15年经验的教师。他曾做过技术员、项目经理、程序员和Web开发人员。他目前的项目中有一个是开发农作物分析工具，这是温室虫害集成管理系统的一个组成部分。他对生物与技术的交叉领域具有强烈的兴趣，并且坚信智能机可以帮助人们解决世界上最重要的问题。

Vahid Mirjalili拥有密歇根州立大学机械工程专业的博士学位，他利用动力学模型开发出模拟蛋白质结构细节的新方法。结合自己在统计学、数据挖掘与物理学领域的知识，他设计了一种基于数据驱动的强大方法，这使得其研究团队在2012年和2014年两次赢得世界性的蛋白质结构预测技术评估（CASP）竞赛。

在攻读博士学位期间，他决定加入到密歇根州立大学计算机科学与工程系，专门从事机器学习领域的研究。目前的研究项目包括海量数据的无监督机器学习算法等。他还是一位充满激情的Python程序员，并且在个人网站http://vahidmirjalili.com上分享了他对聚类算法的实现。

Hamidreza Sattari是IT领域的专业人士，曾参与过开发、架构和管理

等软件工程中的各个环节。他拥有伊朗德黑兰阿萨德大学电气工程专业的学士学位，以及英国海威瓦特大学的软件工程硕士学位。近年来的研究领域是大数据与机器学习。曾与人合著Spring Web Service 2 Cookbook一书，其个人博客地址是：http://justdeveloped-blog.blogspot.com/。

Dmytro Taranovsky是一名软件工程师，对Python、Linux和机器学习感兴趣，并且在这些领域有一定背景。他出生于乌克兰的基辅，1996年移居到美国生活。他从小就表现出了对科学知识的热爱，并曾在数学与物理竞赛中获奖。1999年，被选为美国物理竞赛小组的成员。2005年，他毕业于麻省理工学院的数学专业。此后，担任医疗信息计算机辅助转换（eScription）文本转换系统的软件工程师。虽然最初工作中使用的是Perl，但他更加中意于Python的强大功能和代码的清晰性，Python使得系统轻易扩展到具备处理海量数据的能力。后来他在算法交易公司担任软件工程师与分析师等职务。他在数学基础领域也做出过重大贡献，包括提出了扩展集理论及其与大基数公理之间的关系，提出了真值构建的符号系统，还用Python实现了一个有序符号系统。他享受阅读，喜欢户外运动，希望人们能够生活在一个更加美好的世界中。

前言

无需多言，大家都已知道，机器学习已发展成为当前最能激发人们兴趣的技术之一。出于各种考虑，谷歌、脸书、苹果、亚马逊、IBM等众多大公司都投入了巨资用于机器学习理论和应用的研究。机器学习看起来已经成为当前的一个流行语，但这绝不是炒作。这一令人兴奋的技术为我们带来了全新的可能，并已成为我们日常生活中不可或缺的一部分。例如，与智能手机的语音助手对话、向客户推荐合适的商品、防止信用卡诈骗、过滤垃圾邮件，以及检测与诊断疾病等，这样的例子不胜枚举。

如果你想参与机器学习的实践，或是成为解决问题的能手，抑或是考虑从事机器学习研究方面的工作，那么本书正适合你。不过，对初学者来说，机器学习的理论知识是比较有难度的。幸运的是，近年来出版了许多非常实用的书籍，通过实现一些功能强大的算法来帮助读者步入机器学习的殿堂。在我看来，代码示例起到了重要的作用，通过示例代码的实际操作可以对概念进行更好的阐释。不过请记住：能力强大了，责任就接踵而至！机器学习背后的概念美妙且重要，就如同黑盒一样令人无法琢磨。因此，我旨在为读者提供一本不一样的书籍：讨论与机器学习概念相关的必要细节，并以直观且详实的方式来说明机器学习算法是如何工作的，以及如何在实际中应用它们。尤为重要的是，如何避开常见的误区。

如果在“谷歌学术”中搜索“机器学习”一词，会返回一个大的文献数：1800000。当然，我们无法讨论过去60年中所提出的算法和应用的全部细节。不过，本书涵盖了机器学习领域最核心的主题和概念，可以让大家率

先踏入这一领域，从而开启一段令人兴奋的旅程。如果本书内容无法满足你对此领域的求知欲，你可以利用作者所列出的丰富资源，追寻这一领域最核心的突破。

如果你已经详细地研究过机器学习理论，本书将教会你如何把所学知识付诸实践。如果你曾经使用过机器学习技术，并且希望能更深入地了解机器学习算法是如何工作的，本书也适合你！如果你刚接触机器学习领域，也不用担心，反而更应该感到高兴。我保证，机器学习将改变你解决问题的思考方式，并且让你见识到如何通过发挥数据的力量来解决问题。

在深度进入机器学习领域之前，先回答一个重要的问题：为什么使用Python？答案很简单：它功能强大且使用方便。Python已经成为数据科学领域最为流行的编程语言，它不仅可以让我们忽略掉编程中那些繁杂的部分，还可以提供一个交互式环境，让我们能够快速记录自己的想法，并且将概念直接付诸实现。

回顾我的个人经历，实事求是地说，对机器学习的研究让我成为一名优秀的科学工作者，让我变得善于思考，并且成长为问题解决能手。在本书中，我将与读者分享这些知识。知识经由学习获得，而学习热情是其中的关键，真正掌握某项技能只有通过实践才能够实现。学习的历程不可能一帆风顺，某些主题相对来说难度会比较大，但我希望大家能把握住这个机会，并享受学习的回报。请记住，在机器学习的旅途中，我们共同前行，通过此书，你将学会许多新的技能，借助于它们，我们甚至能够以数据驱动的方式解决那些最棘手的问题。

本书内容

第1章介绍了机器学习算法的划分。此外，还讨论了构建一个典型的机器学习模型的必要步骤，对此过程的理解有助于后续章节的学习。

第2章追溯了机器学习的起源，介绍了二元感知分类器和自适应线性神经元。该章还介绍了模式分类的基础，并着重介绍优化算法和机器学习的相互作用。

第3章介绍了机器学习中分类算法的基本内容，并使用一个流行且包含算法种类相对齐全的开源机器学习算法库scikit-learn，来完成几个机器学习分类算法实例。

第4章讨论了如何处理原始数据中常见的问题，诸如数据缺失等。该章还讨论了几种如何从数据集中找出蕴含信息最丰富特征的方法，并讲解了如何处理不同数据类型的变量，以使其与机器学习算法的输入要求相匹配。

第5章介绍了如何通过降维来压缩数据的特征数量，以便将数据集压缩到一个容量相对较小的子集上，同时还能保持原数据中有用的区分信息。该章主要讨论了通过主成分分析来降维的标准方法，并将其与监督以及线性变换技术进行了比较。

第6章讨论了对预测模型进行评估时应该做什么和不应该做什么。此外，还探讨了衡量模型的不同标准以及机器学习算法调优的相关技巧。

第7章介绍了有效地组合多个机器学习算法的不同方法，并讲授了如

何通过构造继承模型来消除个别分类器的弱点，从而最终得到更加准确和可信的预测结果。

第8章讨论了将文本数据转化为有意义的表达方式的方法，根据人们的留言借助机器学习算法来预测其观点。

第9章继续对预测模型进行了探讨，并将引导你完成将机器学习模型嵌入到Web应用中的核心步骤。

第10章讨论了当自变量和因变量都为连续值时，建立线性模型来完成预测的基本方法。在介绍了不同的线性模型后，又讨论了多项式回归和基于树的方法——随机森林。

第11章将关注的焦点转移到了另一类型的机器学习方法：无监督学习。我们使用了三种基本的聚类算法将具备一定相似度的对象划分为几个组别。

第12章扩展了第2章中提及的梯度优化的概念，并基于流行的反向传播算法来构建强大的多层神经网络。

在上一章内容的基础上，第13章提供了更加有效地训练神经网络的实用指南。该章关注的重点是Theano，它是一个开源的Python库，借此，我们可以充分利用现代GPU上众多的计算内核。

阅读须知

执行本书中的示例代码需要Python 3.4.3或者更高版本，所需操作系

统为Mac OS X、Linux或者微软的Windows。书中的代码需要频繁使用Python科学计算的核心库，包括SciPy、NumPy、scikit-learn、matplotlib和pandas。

第1章将介绍如何设置Python环境和上述核心库，同时还会给出一些有用的技巧。在后续章节中，还会根据需要安装其他库：用于自然语言处理的NLTK库（第8章），Flask Web框架（第9章），统计数据可视化库seaborn（第10章），以及Theano库（第13章），它可用于高效训练图像处理的神经网络单元。

第1章　赋予计算机学习数据的能力

在我看来，机器学习是一门能够发掘数据价值的算法与应用，它是计算机科学中最激动人心的一个领域。我们生活在一个数据资源非常丰富的时代，通过机器学习领域中的自学习算法，可以将这些数据转换为知识。借助于近年来发展起来的诸多功能强大的开源库，现在是进入机器学习领域的最佳时机，同时，利用该领域中那些功能强大的算法来发现数据中的模式，进而实现对未知事件的预测。

本章我们将了解机器学习的主要概念及其几种不同类型的学习算法。同时，还将对相关术语做基本的介绍，为今后熟练使用机器学习解决实际问题打好坚实的基础。

本章将涵盖如下内容：

·机器学习的一般概念

·机器学习方法的三种类型和基本术语

·成功构建机器学习系统所需的模块

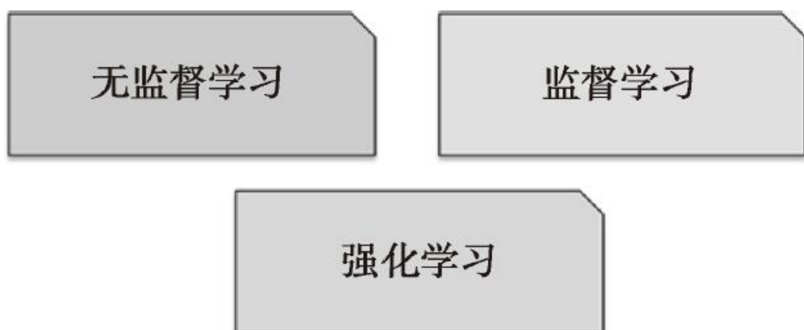
·面向数据分析和机器学习的Python安装与设置

1.1 构建智能机器将数据转化为知识

我们正处于现代技术飞速发展的时代，同时还拥有大量的结构化和非结构化的数据资源。20世纪下半叶，机器学习逐渐演化为人工智能的一个分支，其目的是通过对自学习算法的开发，从数据中获取知识，进而对未来进行预测。与以往通过大量数据分析而人工推导出规则并构造模型不同，机器学习提供了一种从数据中获取知识的方法，同时能够逐步提高预测模型的性能，并将模型应用于基于数据驱动的决策中去。机器学习不仅在计算机科学研究领域显现出日益重要的地位，而且在日常生活中也逐渐发挥出了越来越大的作用。机器学习技术的存在，使得人们可以享受强大的垃圾邮件过滤带来的便利，拥有方便的文字和语音识别软件，能够使用可靠的网络搜索引擎，同时在象棋的网络游戏对阵中棋逢对手，而且在可见的将来，我们将拥有安全高效的无人驾驶汽车。

1.2　机器学习的三种不同方法

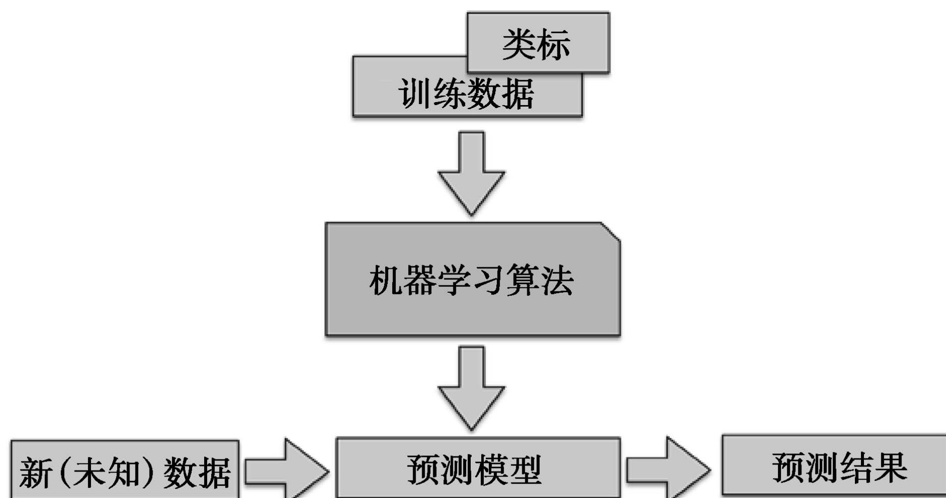
本节我们将介绍三种不同类型的机器学习方法：监督学习（supervised learning）、无监督学习（unsupervised learning）和强化学习（reinforcement learning）。我们将学习三种不同方法之间的本质区别，并使用概念性示例，让大家对这三种方法在实际问题中的应用有一个直观的认识。



1.2.1　通过监督学习对未来事件进行预测

监督学习的主要目的是使用有类标的训练（training）数据构建模型，我们可以使用经训练得到的模型对未来数据进行预测。此处，术语监督（supervised）是指训练数据集中的每个样本均有一个已知的输出项（类标（label））。

以（过滤）垃圾邮件为例：基于有类标的电子邮件样本库，可以使用监督学习算法训练生成一个判定模型，用来判别一封新的电子邮件是否为垃圾邮件；其中，在用于训练的电子邮件样本库中，每一封电子邮件都已被准确地标记是否为垃圾邮件。监督学习一般使用离散的类标（class label），类似于过滤垃圾邮件的这类问题也被称为分类（classification）。监督学习的另一个子类是回归（regression），回归问题的输出项是连续值。

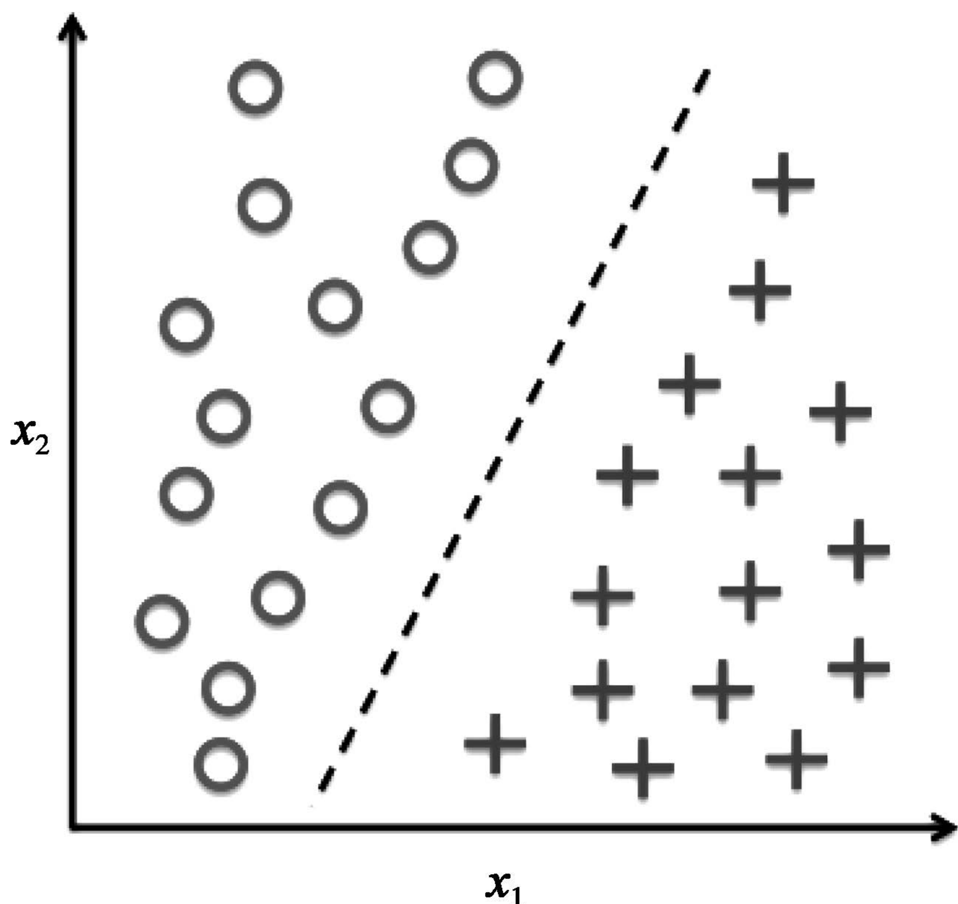


1.利用分类对类标进行预测

分类是监督学习的一个子类，其目的是基于对过往类标已知示例的观察与学习，实现对新样本类标的预测。这些类标是离散的、无序的值，它们可以视为样本的组别信息（group membership）。前面提到的检测垃圾邮件的例子是一个典型的二类别分类（binary classification）任务，机器学习算法会生成一系列的规则用以判定邮件属于垃圾邮件还是非垃圾邮件。

然而，类标集合并非一定是二类别分类的。通过监督学习算法构造的预测模型可以将训练样本库中出现的任何类标赋给一个尚未被标记的新样本。手写字符识别就是一个典型的多类别分类（multi-class classification）的例子。在此，我们可以将字母表中每个字母的多个不同的手写样本收集起来作为训练数据集。此时，若用户通过输入设备给出一个新的手写字符，我们的预测模型能够以一定的准确率将其判定为字母表中的某个字母。然而，如果我们的训练样本库中没有出现0～9的数字字符，那么模型将无法正确辨别任何输入的数字。

下图通过给出具有30个训练样本的实例说明二类别分类任务的概念：15个样本被标记为负类别（negative class）（图中用圆圈表示）；15个样本被标记为正类别（positive class）（图中用加号表示）。此时，我们的数据集是二维的，这意味着每个样本都有两个与其关联的值：x_1和x_2。现在，我们可以通过有监督的机器学习算法获得一条规则，并将其表示为一条黑色虚线标识的分界线、它可以将两类样本分开，并且可以根据给定的x_1、x_2值将新样本划分到某个类别中。



2.使用回归预测连续输出值

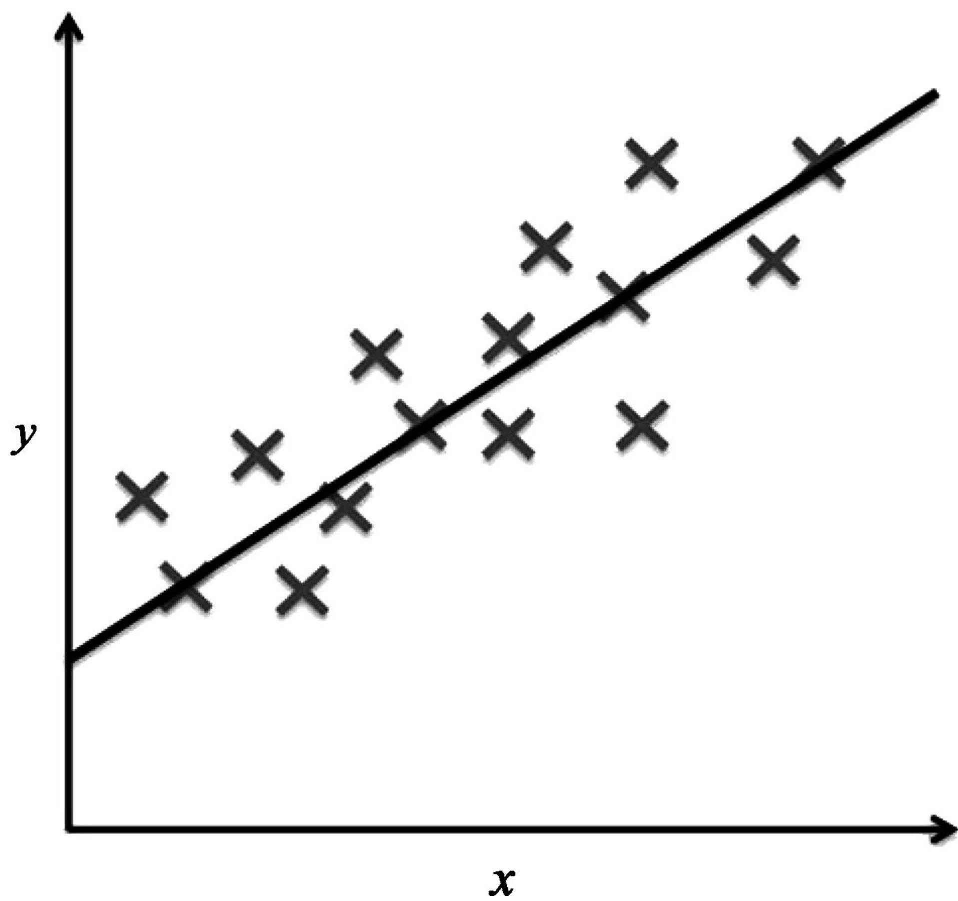
通过上一节的学习，我们知道了分类的任务就是将具有类别的、无序类标分配给各个新样本。另一类监督学习方法针对连续型输出变量进行预测，也就是所谓的回归分析（regression analysis）。在回归分析中，数据中会给出大量的自变量（解释变量）和相应的连续因变量（输出结果），通过尝试寻找这两种变量之间的关系，就能够预测输出变量。

例如，假定我们想预测学生SAT考试中数学科目的成绩。如果花费在

学习上的时间和最终的考试成绩有关联，则可以将其作为训练数据来训练模型，以根据学习时间预测将来要参加考试的学生的成绩。

“回归”一词最早是由Francis Galton于1886年在其论文“Regression Towards Mediocrity in Hereditary Stature”中提出来的。他在文章中描述了这样一种生物现象，即人口身高的方差没有随着时间的推移而增加。他发现：父辈并未将其身高直接遗传给子辈，而子辈的身高更接近于人们的平均身高。

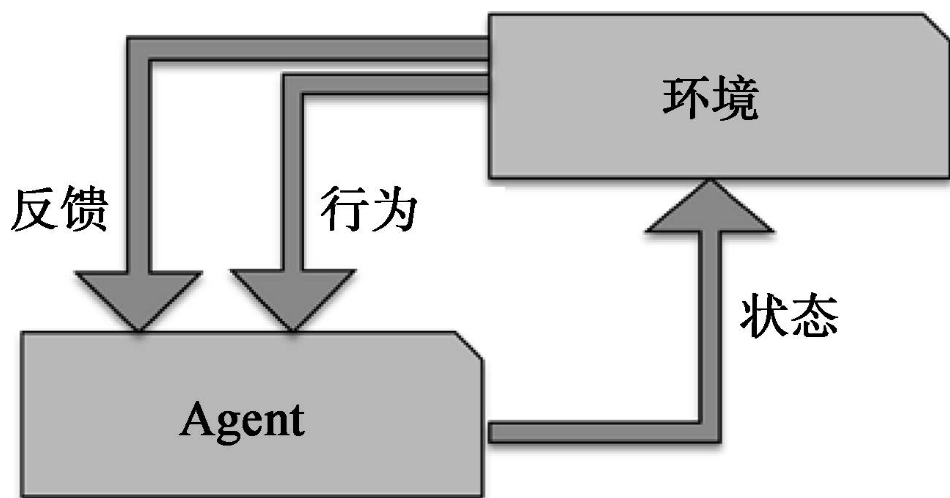
下面用图例阐述了线性回归（linear regression）的概念。给定一个自变量x和因变量y，拟合一条直线使得样例数据点与拟合直线之间的距离最短，最常用的是采用平均平方距离来计算。这样我们就可以通过对样本数据的训练来获得拟合直线的截距和斜率，从而对新的输入变量值所对应的输出变量值进行预测。


y
x

1.2.2 通过强化学习解决交互式问题

另一类机器学习方法是强化学习。强化学习的目标是构建一个系统（Agent），在与环境（environment）交互的过程中提高系统的性能。环境的当前状态信息中通常包含一个反馈（reward）信号，我们可以将强化学习视为与监督学习相关的一个领域。然而，在强化学习中，这个反馈值不是一个确定的类标或者连续类型的值，而是一个通过反馈函数产生的对当前系统行为的评价。通过与环境的交互，Agent可以通过强化学习来得到一系列行为，通过探索性的试错或者借助精心设计的激励系统使得正向反馈最大化。

一个常用的强化学习例子就是象棋对弈的游戏。在此，Agent根据棋盘上的当前局态（环境）决定落子的位置，而游戏结束时胜负的判定可以作为激励信号。



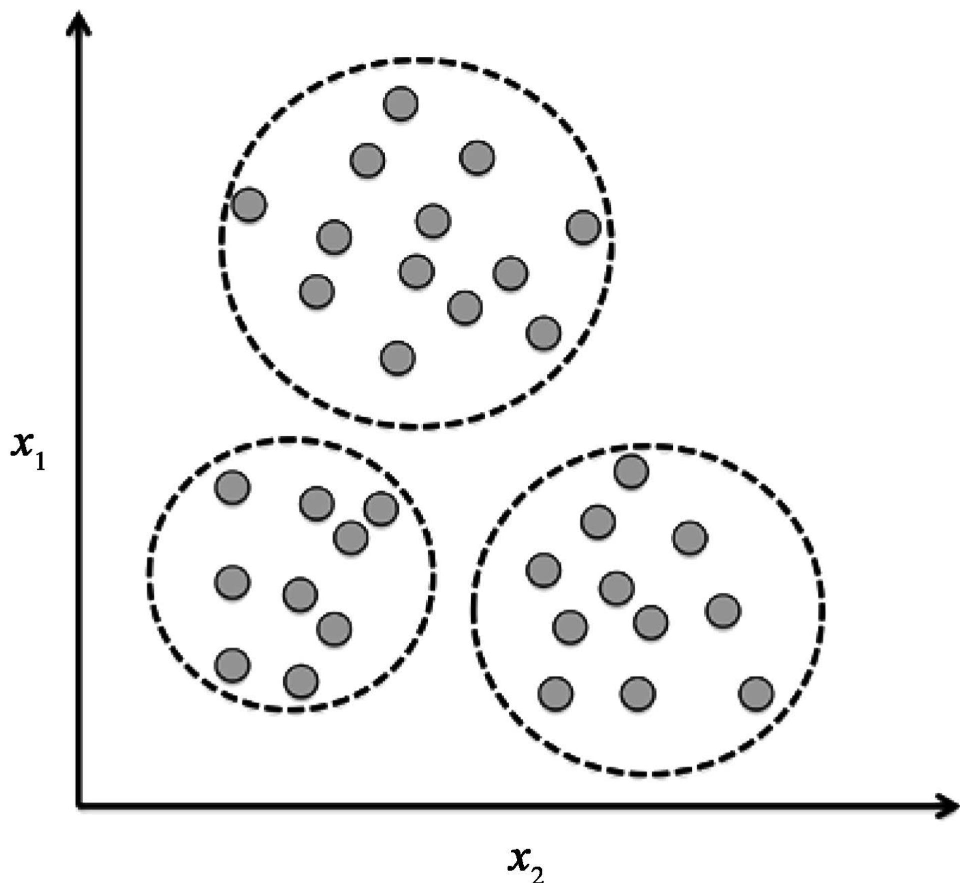
1.2.3　通过无监督学习发现数据本身潜在的结构

在监督学习中，训练模型之前，我们事先可以获知各训练样本对应的目标值。在强化学习中，可以由Agent定义反馈函数对特定行为进行判定。然而，在无监督学习中，我们将处理无类标数据或者是总体分布趋势不明朗的数据。通过无监督学习，我们可以在没有已知输出变量和反馈函数指导的情况下提取有效信息来探索数据的整体结构。

1.通过聚类发现数据的子群

聚类是一种探索性数据分析技术。在没有任何相关先验信息的情况下，它可以帮助我们将数据划分为有意义的小的组别（即簇（cluster））。对数据进行分析时，生成的每个簇中其内部成员之间具有一定的相似度，而与其他簇中的成员则具有较大的不同，这也是为什么聚类有时被称为“无监督分类”。聚类是获取数据的结构信息，以及导出数据间有价值的关系的一种很好的技术，例如，它使得市场人员可以基于用户的兴趣将其分为不同的类别，以分别制定相应的市场营销计划。

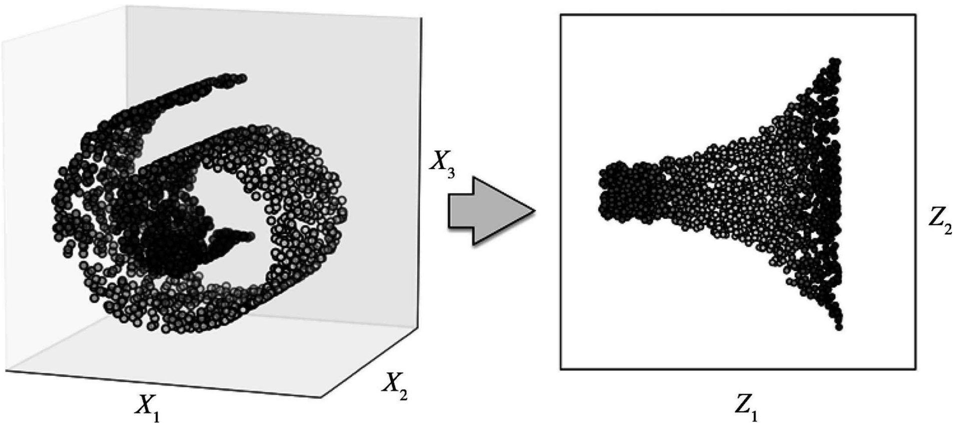
下图演示了聚类方法如何根据数据的x1及x2两个特征值之间的相似性将无类标的数据划分到三个不同的组中。



2.数据压缩中的降维

数据降维（dimensionality reduction）是无监督学习的另一个子领域。通常，我们面对的数据都是高维的（每一次采样都会获取大量的样本值），这就对有限的数据存储空间以及机器学习算法性能提出了挑战。无监督降维是数据特征预处理时常用的技术，用于清除数据中的噪声，它能够在最大程度保留相关信息的情况下将数据压缩到一个维度较小的子空间，但同时也可能会降低某些算法在准确性方面的性能。

降维技术有时在数据可视化方面也是非常有用的。例如，一个具有高维属性的数据集可以映射到一维、二维或者三维的属性空间，并通过三维或二维的散点图和直方图对数据进行可视化展示。下图展示了一个使用非线性降维方法将三维的Swiss Roll压缩到二维特征子空间的实例。



1.2.4 基本术语及符号介绍

目前我们已经讨论了机器学习的三大方法：监督学习、无监督学习和强化学习。在此，我们介绍一下下一章将要用到的一些基本术语。下面表格摘录了鸢尾花数据集（Iris dataset）中的部分数据，鸢尾花数据集是机器学习领域的一个经典示例，它包含了Setosa、Versicolor和Virginica三个品种总共150种鸢尾花的测量数据。其中，每一个样本代表数据集中的一行，而花的测量值以厘米为度量单位存储为列，我们将其定义为数据集的特征。

样本（样例、实际观察到的数据）

花瓣

	萼片长度	萼片宽度	花瓣长度	花瓣宽度	类标
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

萼片

类标（目标数据）

特征（属性、度量、维度）

为了保证描述过程中所用符号及推理过程简单、高效，我们将采用线性代数（Linear algebra）中的一些基本知识。在后续章节中，我们将主要

使用矩阵和向量来标识数据。并做如下约定：矩阵X中的每一行代表一个样本，而样本中的每个特征都表示为单独的列。

在鸢尾花数据集中，包含150个样本和4个特征，因此将其记作150×4维的矩阵$X \in R^{150 \times 4}$：

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

在本书中，我们将使用上标（i）来指代第i个训练样本，使用下标（j）来指代训练数据集中的第j维特征。

加粗的小写字母代表向量（$\boldsymbol{x} \in \mathbb{R}^{n \times 1}$），而加粗的大写字母则代表矩阵（$\boldsymbol{X} \in \mathbb{R}^{n \times n}$）。斜体的 $x^{(n)}$ 和 x_m^n 分别代表向量、矩阵中的单个元素。

例如：x_1^{150}为第 150 个花朵样本的第 1 个特征“萼片宽度”。在这样一个特征矩阵中，每一行代表一个花朵的样本，可记为一个四维行向量 $\boldsymbol{x}^{(i)} \in \mathbb{R}^{1 \times 4}$，$\boldsymbol{x}^{(i)} = [x_1^{(i)} \quad x_2^{(i)} \quad x_3^{(i)} \quad x_4^{(i)}]$。

数据集中的每个特征可以看作是一个 150 维的列向量 $\boldsymbol{x}_{(j)} \in \mathbb{R}^{150 \times 1}$，

$$\boldsymbol{x}_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}$$

类似地，可以用一个150维的列向量来存储目标变量（在本例中为类标）：

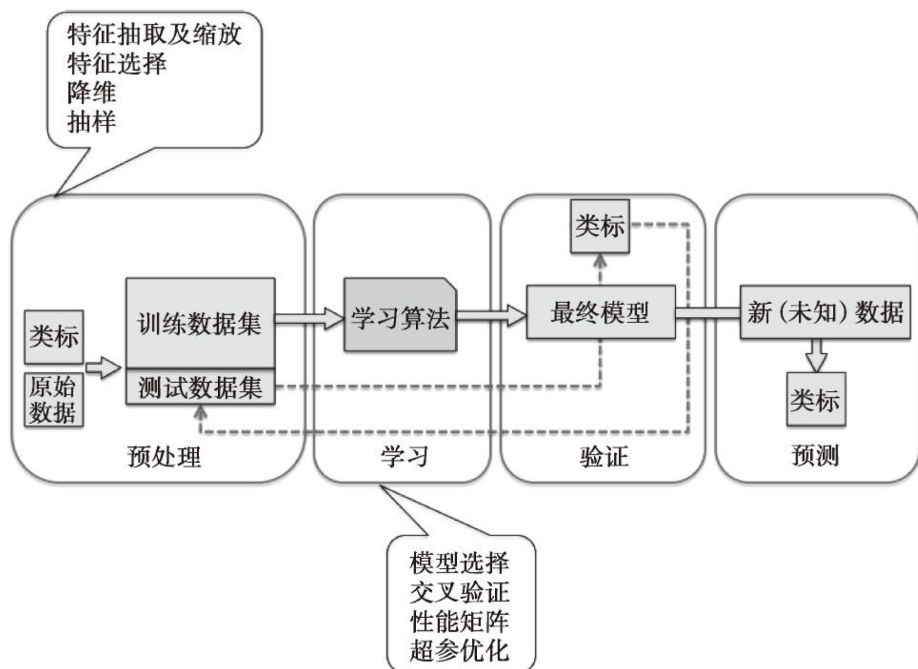
$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(150)} \end{bmatrix} (y \in \{\text{Setosa, Versicolor, Virginica}\})$$

1.3　构建机器学习系统的蓝图

我们已经在前序章节中讨论了机器学习的基本概念和三种不同类型的机器学习方法。在本小节，我们将讨论其他与机器学习系统和学习算法相关的重要内容。下图展示了使用机器学习的预测模型进行数据分析的典型工作流程图，这些内容将在后续小节中做进一步讨论。

1.3.1 数据预处理

为了尽可能发挥机器学习算法的性能，往往对原始数据的格式等有一些特定的要求，但原始数据很少能达到此标准。因此，数据预处理是机器学习应用过程中必不可少的重要步骤之一。以前面提及的鸢尾花数据集为例，我们可以将花朵的图像看作原始数据，从中提取有用的特征。有效的特征可以是花的颜色、饱和度、色彩强度，花朵的整体长度，以及花冠的长度和宽度等。许多机器学习算法为达到性能最优的目的，将属性值映射到[0,1]区间，或者使其满足方差为1、为值为0的标准正态分布，从而使得提取出的特征具有相同的度量标准。我们将在后续章节中对此做进一步的讨论。



某些属性间可能存在较高的关联，因此存在一定的数据冗余。在此情况下，使用数据降维技术将数据压缩到相对低维度的子空间是非常有用的。数据降维不仅能够使得所需的存储空间更小，而且还能够使学习算法运行得更快。

为了保证算法不仅在训练数据集上有效，同时还能很好地应用于新数据，我们通常会随机地将数据集划分为训练数据集和测试数据集。我们使用训练数据集来训练及优化我们的机器学习模型，在完成后，使用测试数据集对最终的模型进行评估。

1.3.2 选择预测模型类型并进行训练

后续章节中我们将会看到，目前已经有多种不同的机器学习算法用来解决不同的问题。从David Wolpert的著名论断“天下没有免费的午餐”中，我们可以总结出重要的一点：我们无法真正免费使用学习算法[1]。直觉上，我们会联想到另外一个流行的谚语，“如果你手上有一把锤子，你会把所有的问题都看作是钉子，我觉得这个有一种莫名的吸引力”（Abraham Maslow，1996）。举例来说：任何分类算法都有其内在的局限性，如果我们不对分类任务预先做一些设定，没有任何一个分类模型会比其他模型更具优势。因此在实际解决问题过程中，必不可少的一个环节就是选用几种不同的算法来训练模型，并比较它们的性能，从中选择最优的一个。在比较不同模型之前，我们需要先确定一个指标来衡量算法的性能。一个常用的指标就是分类的准确率，指的是被正确分类的样例所占的比例。

正常情况下大家会问：我们在选择训练模型的时候没有使用测试数据集，却将这些数据用于最终的模型评估，那该如何判断究竟哪一个模型会在测试数据集和实测数据上有更好的表现？针对该问题，我们可以采用交叉验证技术，将训练数据集进一步分为训练子集和验证子集，从而对模型的泛化能力进行评估。不同机器学习算法的默认参数对于特定类型的任务来说，肯定不是最优的。后续章节将介绍如何使用超参优化技术来帮助我们提升训练模型的性能。直观上看，超参并不是从数据集中训练得出的参数，但却是我们借以提高模型性能的关键节点。后续章节将结合实例对这些内容进行更清晰的介绍。

[1] The Lack of A Priori Distinctions Between Learning Algorithms,D.H.Wolpert 1996;No Free Lunch Theorems for Optimization,D.H.Wolpert and W.G.Macready,1997.

1.3.3 模型验证与使用未知数据进行预测

在使用训练数据集构建出一个模型之后，可以采用测试数据集对模型进行测试，预测该模型在未知数据上的表现并对模型的泛化误差进行评估。如果我们对模型的评估结果表示满意，就可以使用此模型对以后新的未知数据进行预测。有一点需要注意，之前所提到的特征缩放、降维等步骤中所需的参数，只可以从训练数据集中获取，并能够应用于测试数据集及新的数据样本，但仅在测试集上对模型进行性能评估或许无法侦测模型是否被过度优化。

1.4 Python在机器学习中的应用

Python是数据科学领域最流行的编程语言之一，因此拥有大量由众多社区开发的附加扩展库。

对于计算密集型任务，尽管解释型语言（如Python）在性能方面不如低级别语言，但使用相对低级别语言（如Fortran和C等）开发的扩展库（如NumPy、SciPy等）实现了多维数组高速向量化的运算。

处理机器学习程序开发任务，我们主要使用最流行的开源机器学习库scikit-learn来完成。

安装Python包

Python可用于主流的三大操作系统：Microsoft Windows、Mac OS X和Linux。所有版本的安装程序、文档均可以从Python官网下载：https://www.python.org。

本书中示例需使用Python 3.4.3及以上版本，建议读者安装Python 3的最新版本，不过大部分示例程序均兼容Python 2.7.10及之后的版本。如果读者决定使用Python 2.7运行示例代码，请确保了解这两个Python版本之间的主要区别。链接https://wiki.python.org/moin/Python2orPython3详细地比较了Python 3.4和2.7之间的差异。

本书中所用到的其他包可通过pip来进行安装，在Python 3.3中，pip已经默认为Python标准库的一个组成部分。更多信息请参照链接：

https://docs.python.org/3/installing/index.html。

成功安装Python后，我们可以在终端中通过pip命令安装附加Python包：

```
pip install SomePackage
```

已经安装的扩展包，可通过——upgrade选项对其进行更新：

```
pip install SomePackage --upgrade
```

强烈推荐由Continuum Analytics开发的Python版本Anaconda来进行科学计算。Anaconda是一款免费的、内置商业应用的Python版本，它已经内置了应用于数据科学、数学、工程领域所需的核心包，是一个用户友好的跨平台发行版本。Anaconda的安装程序可通过链接http://continuum.io/downloads#py34下载，链接https://store.continuum.io/static/img/Anaconda-Quickstart.pdf提供其快速指南手册。

在成功安装Anaconda后，我们可以使用如下命令安装Python包：

```
conda install SomePackage
```

可使用如下命令更新现有包：

```
conda update SomePackage
```

本书主要使用NumPy的多维数组存储和处理数据。我们也会用到pandas，它是一个建立在NumPy上、更方便处理表格类数据的附加工具包。为了使读者对数据有个直观的感觉，以提高学习体验和数据可视化的质量，我们使用了可高度定制的matplotlib库。

本书使用的主要的Python包的版本如下。请确保你选择的版本号满足此要求或者比所列版本更新，以保证示例代码能正确运行。

·NumPy 1.9.1

·SciPy 0.14.0

·scikit-learn 0.15.2

·matplotlib 1.4.0

·pandas 0.15.2

本章小结

在本章中，我们从宏观上讨论了机器学习，并且介绍相关的重点和主要概念，后续章节会对这些问题进行更详细的探讨。

通过本章的学习，我们已经了解到，监督学习由两个重要的子领域组成：分类和回归。其中，我们可以通过分类技术将对象划分到不同的类别中，而回归则能够对输出为连续型的目标变量进行预测。无监督学习不仅能从众多的无类标数据中发现其整体结构，同时在特征预处理阶段的数据压缩中也发挥了重要作用。

我们还简要介绍了将机器学习应用到实际工作中的具体路线图，这些内容也为在后续章节中做进一步探讨和实践打好了基础。最后，还介绍了Python运行环境及其安装过程，并讲解了如何更新所需的包以满足机器学习实战的需要。

在下一章中，我们将实现一个分类领域最早提出的机器学习算法，并以此作为第3章的铺垫。在第3章中，将借助于开源的机器学习库scikit-learn来讲解更加高级的机器学习算法。既然机器学习算法是通过数据来进行学习，因此如何将有用的信息输入到算法中是至关重要的，第4章将介绍数据预处理技术的几种重要方法。降维是数据预处理领域的一种重要技术，它使得我们可以将数据压缩到一个相对低维的特征子空间上，这对提高机器学习算法的计算效率是非常有效的，大家可以在第5章学到降维的相关内容。在第6章，大家将学习到模型评估和参数调优技术。不过，在某些情况下，即使在参数调优与测试上花费了大量的时间与精力，模型的

预测性能可能仍旧无法达到我们的预期。第7章将介绍如何通过组合不同的机器学习算法来构造一个性能更加强大的预测系统。

在掌握了机器学习领域中相关的重要概念之后，我们将了解一下机器学习在实际工作中的应用：第8章介绍使用模型来对文本数据做情感分析；而通过第9章的学习，大家可以了解如何将机器学习模型嵌入到Web应用程序中，使其在整个网络范围内得以共享。如何对目标值为连续变量的数据进行预测分析是机器学习领域的一个重要分支，我们将在第10章中通过回归分析来对其进行讲解。第11章则主要介绍如何通过聚类算法来发现数据中隐含的结构。本书最后将介绍神经网络，这是目前机器学习研究领域的热门话题之一，它使得我们可以解决复杂的问题，如图像和语音识别等。

第2章　机器学习分类算法

本章将介绍最早以算法方式描述的分类机器学习算法：感知器（perceptron）和自适应线性神经元（adaptive linear neuron）。我们将使用Python循序渐进地实现一个感知器，并且通过训练使其具备对鸢尾花数据集中数据进行分类的能力。这将帮助我们理解机器学习算法中分类的概念，以及如何使用Python高效地实现分类算法。通过对自适应线性神经元优化基础的讨论，将会对我们在第3章中使用scikit-learn机器学习库开发更加高效的分类器奠定基础。

本章将涵盖如下内容：

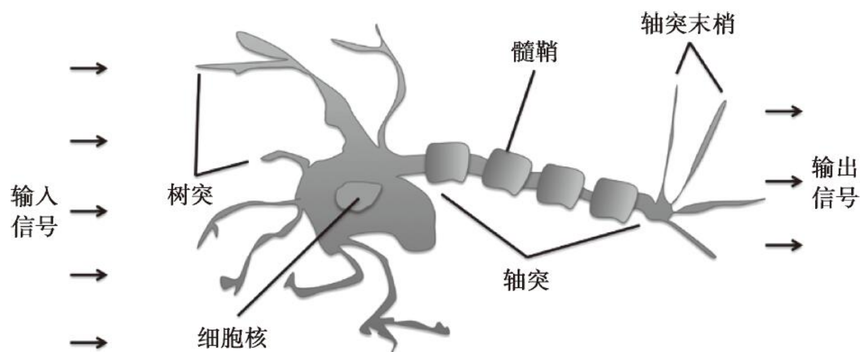
·培养对机器学习算法的直观认识

·使用pandas、NumPy和matplotlib读取、处理和可视化数据

·使用Python实现线性分类算法

2.1 人造神经元——早期机器学习概览

在详细讨论感知器和相关算法之前，先大体了解一下早期机器学习的起源。为了理解大脑的工作原理以设计人工智能系统，沃伦·麦卡洛可（Warren McCullock）与沃尔特·皮茨（Walter Pitts）在1943年提出了第一个脑神经元的抽象模型，也称作麦卡洛可-皮茨神经元（McCullock-Pitts neuron，MCP）[1]，神经元是大脑中相互连接的神经细胞，它可以处理和传递化学和电信号，如下图所示：



麦卡洛可和皮茨将神经细胞描述为一个具备二进制输出的逻辑门。树突接收多个输入信号，如果累加的信号超过某一阈值，经细胞体的整合就会生成一个输出信号，并通过轴突进行传递。

几年后，弗兰克·罗森布拉特（Frank Rossenblatt）基于MCP神经元模型提出了第一个感知器学习法则[2]。在此感知器规则中，罗森布拉特提出了一个自学习算法，此算法可以自动通过优化得到权重系数，此系数与输入值的乘积决定了神经元是否被激活。在监督学习与分类中，类似算法可

用于预测样本所属的类别。

更严谨地讲，我们把这个问题看作一个二值分类的任务，为了简单起见，我们把两类分别记为1（正类别）和-1（负类别）。我们可以定义一个激励函数（activation function）φ（z），它以特定的输入值x与相应的权值向量w的线性组合作为输入，其中，z也称作净输入（$z = w_1x_1 + \ldots + w_mx_m$）：

$$\boldsymbol{w}=\begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix},\ \boldsymbol{x}=\begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

此时，对于一个特定样本$x^{(i)}$的激励，也就是φ（z）的输出，如果其值大于预设的阈值θ，我们将其划分到1类，否则为-1类。在感知器算法中，激励函数φ（·）是一个简单的分段函数。

$$\phi(z)=\begin{cases} 1 & 若\ z \geqslant \theta \\ -1 & 其他 \end{cases}$$

为了简单起见，我们可以把阈值θ移到等式的左边，并增加一个初始项权重记为$w_0=-\theta$且设$x_0=1$，这样我们就可以把z写成一个更加紧凑的形式：$z=w_0x_0+w_1x_1+\cdots+w_mx_m=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$，$\phi(z)=\begin{cases} 1 & 若\ z \geqslant \theta \\ -1 & 其他 \end{cases}$。

注意：在下面的小节中，借用了线性代数的部分符号。例如，使用向量点积（vector dot product）来表示x和w乘积的和，而上标T则表示转置，它使得行向量和列向量之间能够相互转换：

$$z=w_0x_0+w_1x_1+\cdots+\boldsymbol{w}_m\boldsymbol{x}_m=\sum_{j=0}^{m}\boldsymbol{w}_j\boldsymbol{x}_j=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$$

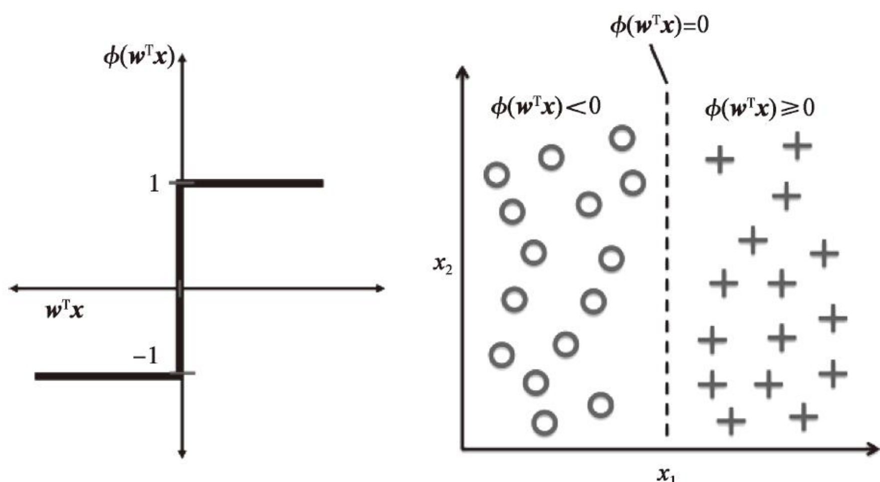
例如：$[1\quad 2\quad 3]\times\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}=1\times4+2\times5+3\times6=32$。

此外，转置操作也可以通过其主对角线应用到矩阵上，例如：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

在本书中，我们仅使用了线性代数中最基本的符号。不过，如果读者想要快速复习一下相关内容，请参考Zico Kolter的线性代数可通过如下链接免费获取：http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf。

下图中，左图说明了感知器模型的激励函数如何将输入$z = w^{T}x$转换到二值输出（-1或1），右图说明了感知器模型如何将两个可区分类别进行线性区分。



MCP神经元和罗森布拉特阈值感知器的理念就是，通过模拟的方式还原大脑中单个神经元的工作方式：它是否被激活。这样，罗森布拉特感知器最初的规则非常简单，可总结为如下几步：

1）将权重初始化为零或一个极小的随机数。

2）迭代所有训练样本$x^{(i)}$，执行如下操作：

（1）计算输出值$\hat{y}$。

（2）更新权重。

这里的输出值是指通过前面定义的单位阶跃函数预测得出的类标，而每次对权重向量中每一权重w的更新方式为：

$$w_j := w_j + \Delta w_j$$

对于用于更新权重w_j的值Δw_j，可通过感知器学习规则计算获得：

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$

其中，η为学习速率（一个介于0.0到1.0之间的常数），$y^{(i)}$为第i个样本的真实类标，$\hat{y}^{(i)}$为预测得到的类标。需特别注意，权重向量中的所有权重值是同时更新的，这意味着在所有的权重Δw_j更新前，我们无法重新计算$\hat{y}^{(i)}$。具体地，对于一个二维数据集，可通过下式进行更新：

$$\Delta w_0 = \eta\left(y^{(i)} - output^{(i)}\right)$$
$$\Delta w_1 = \eta\left(y^{(i)} - output^{(i)}\right)x_1^{(i)}$$
$$\Delta w_2 = \eta\left(y^{(i)} - output^{(i)}\right)x_2^{(i)}$$

在使用Python语言实现感知器之前，让我们通过一个简单的推导实验来体验一下这个学习规则的简洁之美。对于如下式所示的两种场景，若感知器对类标的预测正确，权重可不做更新：

$$\Delta w_j = \eta(-1^{(i)} - (-1^{(i)}))x_j^{(i)} = 0$$
$$\Delta w_j = \eta(1^{(i)} - 1^{(i)})x_j^{(i)} = 0$$

但是，在类标预测错误的情况下，权重的值会分别趋向于正类别或者负类别的方向：

$$\Delta w_j = \eta(1^{(i)} - (-1^{(i)}))x_j^{(i)} = \eta(2)x_j^{(i)}$$
$$\Delta w_j = \eta(-1^{(i)} - 1^{(i)})x_j^{(i)} = \eta(-2)x_j^{(i)}$$

为了对乘法因子$x_j^{(i)}$有个更直观的认识，我们看另一个简单的例子，其中：

$$\hat{y}_j^{(i)} = +1, \ y^{(i)} = -1, \ \eta = 1$$

假定$x_j^{(i)} = 0.5$，且模型将此样本错误地分到了-1类别内。在此情况下，我们应将相应的权值增1，以保证下次遇到此样本时使得激励$x_j^{(i)} = w_j^{(i)}$能将其更多地判定为正类别，这也相当于增大其值大于单位阶跃函数阈值的概率，以使得此样本被判定为＋1类。

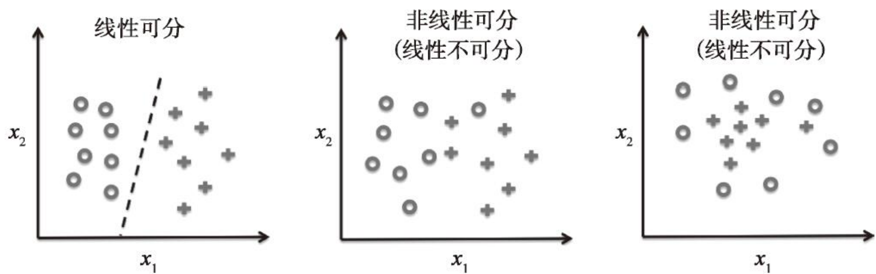
$$\Delta w_j^{(i)} = (1^{(i)} - (-1^{(i)}))0.5^{(i)} = (2)0.5^{(i)} = 1$$

权重的更新与$x_j^{(i)}$的值成比例。例如，如果有另外一个样本$x_j^{(i)} = 2$被错误分到了-1类别中，我们应更大幅度地移动决策边界，以保证下次遇到此样本时能正确分类。

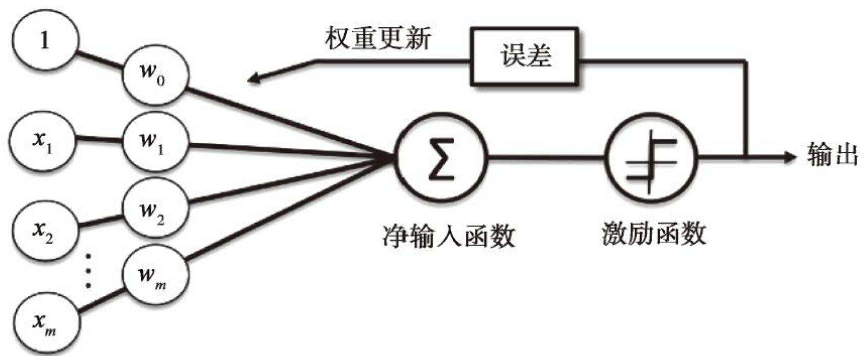
$$\Delta w_j = (1^{(i)} - (-1^{(i)}))2^{(i)} = (2)2^{(i)} = 4$$

需要注意的是：感知器收敛的前提是两个类别必须是线性可分的，且学习速率足够小。如果两个类别无法通过一个线性决策边界进行划分，可以为模型在训练数据集上的学习迭代次数设置一个最大值，或者设置一个

允许错误分类样本数量的阈值——否则，感知器训练算法将永远不停地更新权值。



在学习下一小节中感知器的实现方法之前，我们先通过一个简单的示例图片总结一下刚刚学到的感知器的基本概念：



上图说明了感知器如何接收样本x的输入，并将其与权值w进行加权以计算净输入（net input）。进而净输入被传递到激励函数（在此为单位阶跃函数），然后生成值为+1或者-1的二值输出，并以其作为样本的预测类标。在学习阶段，此输出用来计算预测的误差并更新权重。

[1] W.S.McCulloch and W.Pitts.A Logical Calculus of the Ideas Immanent in Nervous Activity.The bulletin of mathematical

biophysics,5(4):115–133,1943.

[2] F.Rosenblatt,The Perceptron,a Perceiving and Recognizing Automaton.Cornell Aeronautical Laboratory,1957.

2.2　使用Python实现感知器学习算法

在上一节中，我们已经学习了罗森布拉特感知器的工作方式，现在使用Python来实现它，并且将其应用于第1章中提到的鸢尾花数据集中。通过使用面向对象编程的方式在一个Python类中定义感知器的接口，使得我们可以初始化新的感知器对象，并使用类中定义的fit方法从数据中进行学习，使用predict方法进行预测。按照Python开发的惯例，对于那些并非在初始化对象时创建但是又被对象中其他方法调用的属性，可以在后面添加一个下划线，例如：self.w_。

若读者不熟悉Python的科学计算库，或者想对其有更深入的了解，请参见如下资源：

NumPy：http://wiki.scipy.org/Tentative_NumPy_Tutorial。

pandas：http://pandas.pydata.org/pandas-docs/stable/tutorials.html。

matplotlib：http://matplotlib.org/ussers/beginner.html。

此外，为了更好地学习书中的代码，建议读者通过Packt的网站下载与本书相关的IPython notebook文件。关于IPython notebook的使用介绍，请参见链接：https://ipython.org/ipython-doc/3/notebook/index.html。

```
import numpy as np
class Perceptron(object):
    """Perceptron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclassifications in every epoch.

    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples
            is the number of samples and
        n_features is the number of features.
    y : array-like, shape = [n_samples]
        Target values.

    Returns
    -------
```

```
        self : object

        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

在感知器实现过程中，我们实例化一个Perceptron对象时，给出了一个学习速率eta和在训练数据集上进行迭代的次数n_iter。通过fit方法，我们将self.w_中的权值初始化为一个零向量R^{m+1}，其中m是数据集中维度（特征）的数量，我们在此基础上增加一个0权重列（也就是阈值）。

与Python的列表类似，NumPy也使用方括号（[ ]）对一维数组进行索引。对二维数组来说，第1个索引值对应数组的行，第2个索引值对应数组的列。例如，X[2,3]表示二维数组X中第3行第4列所对应的元素。

初始化权重后，fit方法循环迭代数据集中的所有样本，并根据前面章节讨论过的感知器学习规则来更新权重。我们使用predict方法来计算类标，这个方法在fit方法中也被调用，用于计算权重更新时的类标，在完成模型训练后，predict方法也用于预测未知数据的类标。此外，在每次迭代的过程中，我们收集每轮迭代中错误分类样本的数量，并将其存放于列表

self.erros_中，以便后续对感知器在训练中表现的好坏做出判定。另外，在net_input方法中使用的np.dot方法用于计算向量的点积w^Tx。

除了使用NumPy的a.dot（b）或np.dot（a,b）计算两个数组a和b的点积之外，还可以通过类似sum（i*j for i,j in zip（a,b））这样的经典Python for循环结构来计算。与使用Python的循环结构相比，NumPy的优势在于其算术运算的向量化。向量化意味着一个算术运算操作会自动应用到数组中的所有元素上。通过在指令序列中配置所需的算术运算，我们便可充分利用现代处理器单指令流多数据流（Single Instruction,Multiple Data,SIMD）架构的支持快速完成运算，而不是循环地对每个元素逐一进行计算。此外，NumPy还使用了高度优化的线性代数库，如使用C或者Fortran实现的基本线性代数子程序（Basic Linear Algebra Subprogram，BLAS）和线性代数包（Linear Algebra Package，LAPACK）。最后，NumPy还允许我们使用基本的线性代数操作这类加紧凑和直观的方式编写代码，如向量和矩阵的点积。

基于鸢尾花数据集训练感知器模型

为了测试前面实现的感知器算法，我们从鸢尾花数据集中挑选了山鸢尾（Setosa）和变色鸢尾（Versicolor）两种花的信息作为测试数据。虽然感知器并不将数据样本特征的数量限定为两个，但出于可视化方面的原因，我们只考虑数据集中萼片长度（sepal length）和花瓣长度（petal-length）这两个特征。同时，选择山鸢尾和变色鸢尾也是出于实践需要的考虑。不过，感知器算法可以扩展到多类别的分类器应用中，比如通过一对多（One-vs.-all）技术。

一对多（One-vs.-All，OvA），有时也称为一对其他（One-vs.-Rest，OvR），是一种将二值分类器扩充到多类别分类任务上的一种技术。我们可以使用QvA针对每个类别训练一个分类器，其中分类器所对应类别的样本为正类别，其他所有类别的样本为负类别。当应用与新数据样本识别时，我们可以借助于分类器φ（z），其中m为类标数量，并将相关度最高的类标赋给待识别样本。对于感知器来说，就是最大净输入值绝对值对应的类标。

首先，我们使用pandas库直接从UCI机器学习库中将鸢尾花数据集转换为DataFrame对象并加载到内存中，并使用tail方法显示数据的最后5行以确保数据正确加载。

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...     'machine-learning-databases/iris/iris.data', header=None)
>>> df.tail()
```

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	6.9	3.0	5.1	1.8	Iris-virginica

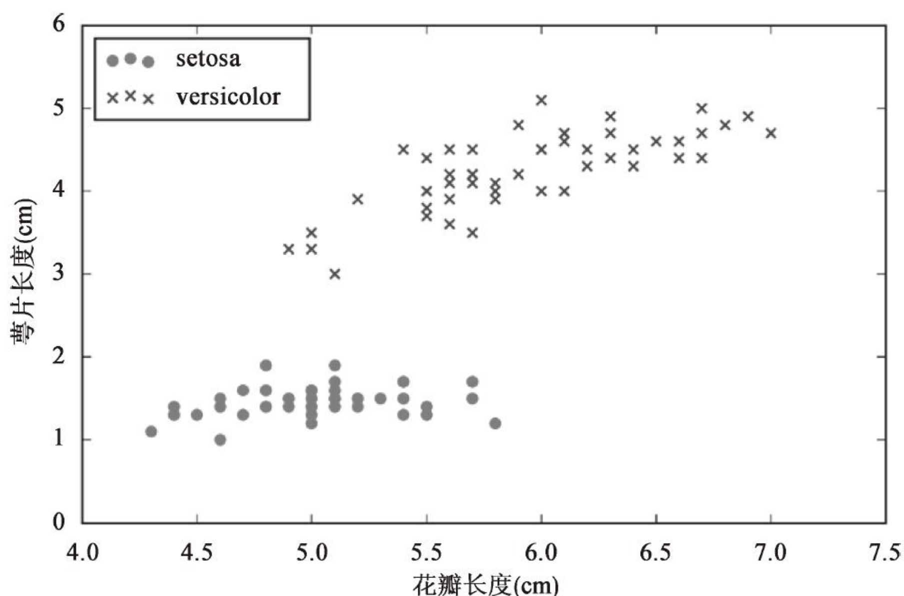
接下来，我们从中提取前100个类标，其中分别包含50个山鸢尾类标和50个变色鸢尾类标，并将这些类标用两个整数值来替代：1代表变色鸢尾，-1代表山鸢尾，同时把pandas DataFrame产生的对应的整数类标赋给NumPy的向量y。类似地，我们提取这100个训练样本的第一个特征列（萼片长度）和第三个特征列（花瓣长度），并赋值给属性矩阵X，这样我们就可以用二维散点图对这些数据进行可视化了。

```
>>> import matplotlib.pyplot as plt
```

```
>>> import numpy as np

>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> X = df.iloc[0:100, [0, 2]].values
>>> plt.scatter(X[:50, 0], X[:50, 1],
...             color='red', marker='o', label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...             color='blue', marker='x', label='versicolor')
>>> plt.xlabel('petal length')
>>> plt.ylabel('sepal length')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

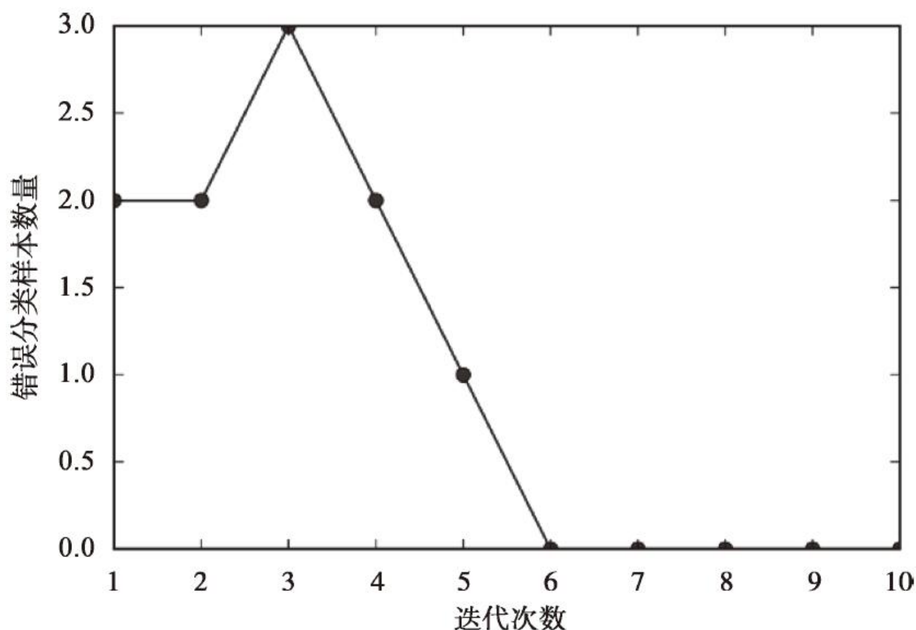
上述代码的执行结果如下图所示：



现在，我们可以利用抽取出的鸢尾花数据子集来训练感知器了。同时，我们还将绘制每次迭代的错误分类数量的折线图，以检验算法是否收敛并找到可以分开两种类型鸢尾花的决策边界。

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_,
...          marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Number of misclassifications')
>>> plt.show()
```

执行上述代码，我们可以看到每次迭代对应的错误分类数量，如下图所示：



如上图所示，我们的分类器在第6次迭代后就已经收敛，并且具备对训练样本进行正确分类的能力。下面通过一个简单的函数来实现对二维数据集决策边界的可视化。

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

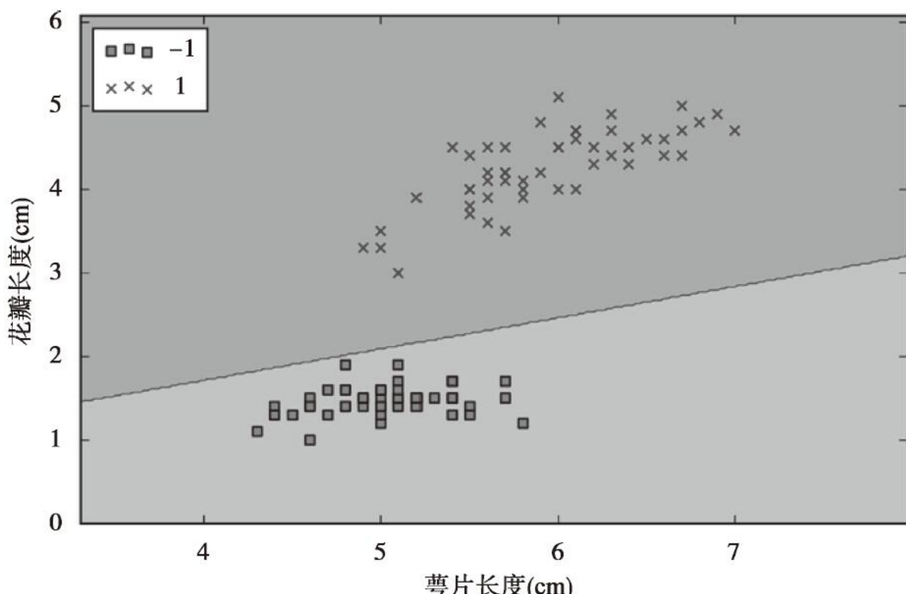
    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)
```

我们先通过ListedColormap方法定义一些颜色（color）和标记符号（marker），并通过颜色列表生成了颜色示例图。然后对两个特征的最大值、最小值做了限定，使用NumPy的meshgrid函数将最大值、最小值向量生成二维数组xx1和xx2。由于使用了两个特征来训练感知器，因此需要将二维组展开，创建一个与鸢尾花数据训练数据集中列数相同的矩阵，以预测多维数组中所有对应点的类标z。将z变换为与xx1和xx2相同的维度后，我们就可以使用matplotlib中的contourf函数，对于网格数组中每个预测的类以不同的颜色绘制出预测得到的决策区域。

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

执行上述代码，我们可以得到决策区域的图像，如下图所示：



正如我们从图中看到的那样，通过感知器学习得到的分类曲面可以完美地对训练子集中的所有样本进行分类。

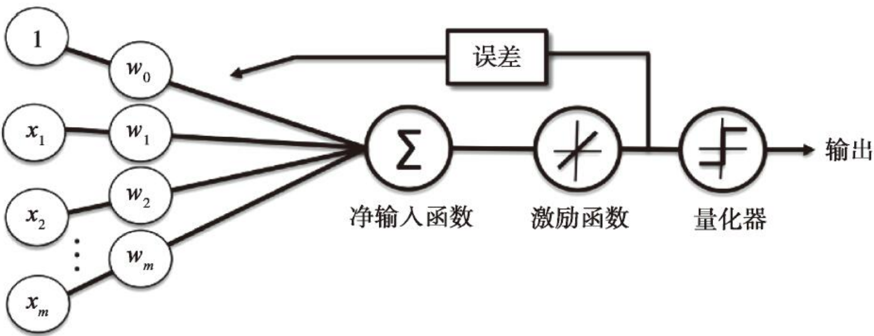
虽然在上例中感知器可以完美地划分两个类别的花，但感知器所面临的最大问题是算法的收敛。Frank Rosenblatt从数学上证明了：如果两个类别可以通过线性超平面进行划分，则感知器算法一定会收敛。但是，如果两个类别无法通过线性判定边界完全正确地划分，则权重会不断更新。为防止发生此类事件，通常事先设置权重更新的最大迭代次数。

2.3　自适应线性神经元及其学习的收敛性

本节将介绍另一种类型的单层神经网络：自适应线性神经网络（Adaptive Linear Neuron，Adaline）。在Frank Rosenblatt提出感知器算法几年之后，Bernard Widrow和他的博士生Tedd Hoff提出了Adaline算法，这可以看作对之前算法的改进。[1]Adeline算法相当有趣，它阐明了代价函数的核心概念，并且对其做了最小化优化，这是理解逻辑斯谛回归（logistic regression）、支持向量机（support vector machine）和后续章节涉及的回归模型等基于回归的高级机器学习分类算法的基础。

基于Adeline规则的权重更新是通过一个连续的线性激励函数来完成的，而不像Rosenblatt感知器那样使用单位阶跃函数，这是二者的主要区别。Adeline算法中作用于净输入的激励函数φ（z）是简单的恒等函数，即φ（w^Tx）＝w^Tx。

线性激励函数在更新权重的同时，我们使用量化器（quantizer）对类标进行预测，量化器与前面提到的单位阶跃函数类似，如下图所示：



将上图与前文介绍的感知器算法的示意图进行比较，可以看出区别在于：这里使用线性激励函数的连续型输出值，而不是二类别分类类标来计算模型的误差以及更新权重。

[1] B.Widrow et al.Adaptive"Adaline"neuron using chemical"memistors".Number Technical Report 1553-2.Stanford Electron.Labs.Stanford，CA，October 1960.

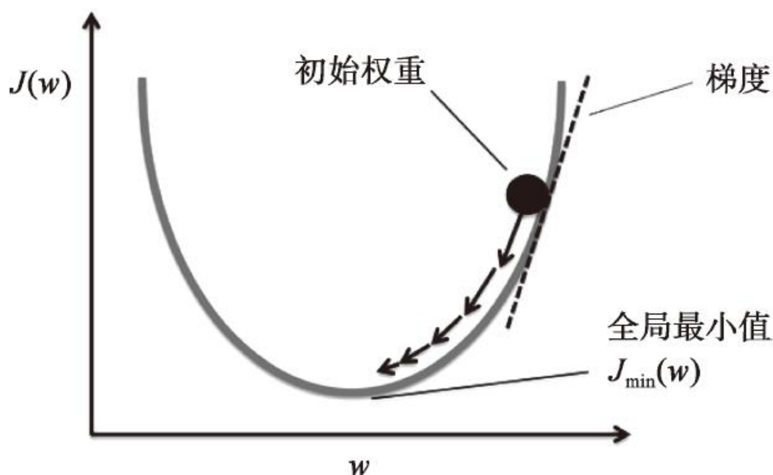
2.3.1 通过梯度下降最小化代价函数

机器学习中监督学习算法的一个核心组成在于：在学习阶段定义一个待优化的目标函数。这个目标函数通常是需要我们做最小化处理的代价函数。在Adaline中，我们可以将代价函数J定义为通过模型得到的输出与实际类标之间的误差平方和（Sum of Squared Error，SSE）：

$$J(w)=\frac{1}{2}\sum_{i}\left(y^{(i)}-\phi\left(z^{(i)}\right)\right)^{2}$$

在此，系数1/2只是出于方便的考虑，它使我们更容易导出梯度，具体将在下一段落中介绍。与单位阶跃函数相比，这种连续线型激励函数的主要优点在于：其代价函数是可导的。此代价函数的另一个优点在于：它是一个凸函数；这样，我们可以通过简单、高效的梯度下降优化算法来得到权重，且能保证在对鸢尾花数据集中样本进行分类时代价函数最小。

如下图所示，我们将梯度下降的原理形象地描述为下山，直到获得一个局部或者全局最小值。在每次迭代中，根据给定的学习速率和梯度的斜率，能够确定每次移动的步幅，我们按照步幅沿着梯度方向前进一步。



通过梯度下降，我们可以基于代价函数J（w）沿梯度▿J（w）方向做一次权重更新：

$$\boldsymbol{w}:=\boldsymbol{w}+\Delta\boldsymbol{w}$$

在此，权重增量Δw定义为负梯度[1]与学习速率η的乘积：

$$\Delta\boldsymbol{w}=-\eta\Delta\mathbf{J}(\boldsymbol{w})$$

为了计算代价函数的梯度，我们需要计算代价函数相对于每个权重 w_j 的偏导$\frac{\partial J}{\partial w_i}=-\sum_i(y^{(i)}-\phi(z^{(i)}))x_j^{(i)}$，这样我们就可以把对权重 w_j 的更新写作 $\Delta\boldsymbol{w}_j=-\eta\frac{\partial J}{\partial w_i}=\mu\sum_i(y^{(i)}-\phi(z^{(i)}))x_j^{(i)}$。由于所有权重被同时更新，Adaline 学习规则可记为：$\boldsymbol{w}=\boldsymbol{w}+\Delta\boldsymbol{w}$。

对于熟悉代数的读者来说，误差平方和代价函数对应于第j个权重的偏导，可通过下列步骤得到：

$$
\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i\left(y^{(i)}-\phi\left(z^{(i)}\right)\right)^2 \\
&= \frac{1}{2}\frac{\partial}{\partial w_j}\sum_i\left(y^{(i)}-\phi\left(z^{(i)}\right)\right)^2 \\
&= \frac{1}{2}\sum_i 2\left(y^{(i)}-\phi\left(z^{(i)}\right)\right)\frac{\partial}{\partial w_j}\left(y^{(i)}-\phi\left(z^{(i)}\right)\right) \\
&= \sum_i\left(y^{(i)}-\phi\left(z^{(i)}\right)\right)\frac{\partial}{\partial w_j}\left(y^{(i)}-\sum_i\left(w_j^{(i)}x_j^{(i)}\right)\right) \\
&= \sum_i\left(y^{(i)}-\phi\left(z^{(i)}\right)\right)\left(-x_j^{(i)}\right) \\
&= -\sum_i\left(y^{(i)}-\phi\left(z^{(i)}\right)\right)x_j^{(i)}
\end{aligned}
$$

虽然Adaline学习规则与感知器规则看起来类似，不过$\varphi(z^{(i)})$中的$z^{(i)}=w^{T}x^{(i)}$是实数，而不是整数类标。此外，权重的更新是基于训练集中所有样本完成的（而不是每次一个样本渐进更新权重），这也是此方法被称作“批量”梯度下降的原因。

[1] 梯度相反的方向是学习速率最快的方向。——译者注

2.3.2 使用Python实现自适应线性神经元

感知器规则与Adaline非常相似，我们将在前面实现的感知器代码的基础上修改fit方法，将其权重的更新改为通过梯度下降最小化代价函数来实现Adaline算法。

```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
```

```
        Weights after fitting.
    errors_ : list
        Number of misclassifications in every epoch.

    """
    def __init__(self, eta=0.01, n_iter=50):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors,
            where n_samples is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
        self : object

        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

与感知器中针对每一个样本做一次权重更新不同，我们基于训练数据

集通过self.eta*errors.sum()来计算梯度的第0个位置的权重，通过self.eta*X.T.dot（errors）来计算1到m位置的权重，其中X.T.dot（errors）是特征矩阵与误差向量之间的乘积。与前面感知器的实现类似，我们设置一个列表self.cost_来存储代价函数的输出值以检查本轮训练后算法是否收敛。

矩阵与向量的乘法计算类似于将矩阵中的每一行单独作为一个行向量与原列向量的点积计算。这种向量化方法使得书写更加紧凑，同时借助于NumPy也使得计算更加高效。例如：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 2 \times 8 + 3 \times 9 \\ 4 \times 7 + 5 \times 8 + 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

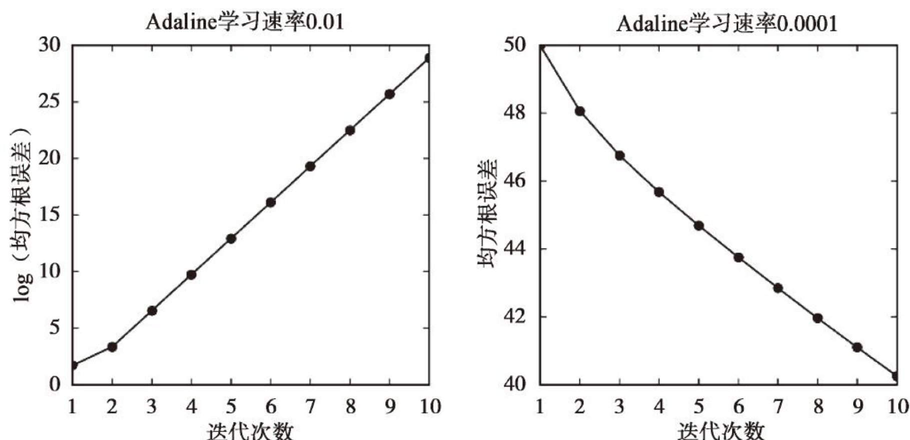
在实践中，为优化收敛效果，常常需要通过实验来找到合适的学习速率η。因此，我们分别使用η = 0.1和η = 0.0001两个学习速率来绘制迭代次数与代价函数的图像，以观察Adaline通过训练数据进行学习的效果。

这里的学习速率η和迭代次数n_iter都被称作感知器和Adaline算法的超参（hyperp-arameter）。在第4章，我们将学习自动调整超参值以得到分类性能最优模型的各种技术。

现在我们分别绘制在两个不同的学习速率下，代价函数与迭代次数的图像。

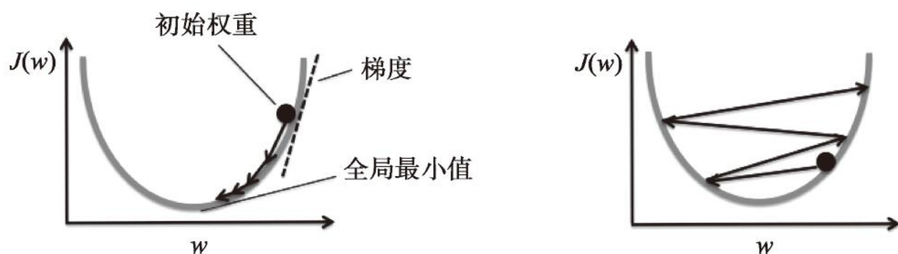
```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
>>> ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.cost_) + 1),
...            np.log10(ada1.cost_), marker='o')
>>> ax[0].set_xlabel('Epochs')
>>> ax[0].set_ylabel('log(Sum-squared-error)')
>>> ax[0].set_title('Adaline - Learning rate 0.01')
>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.cost_) + 1),
...            ada2.cost_, marker='o')
>>> ax[1].set_xlabel('Epochs')
>>> ax[1].set_ylabel('Sum-squared-error')
>>> ax[1].set_title('Adaline - Learning rate 0.0001')
>>> plt.show()
```

从下面代价函数输出结果的图像中可以看到，我们面临两种不同类型的问题。左边的图像显示了学习速率过大可能会出现的问题——并没有使代价函数的值尽可能的低，反而因为算法跳过了全局最优解，导致误差随着迭代次数增加而增大。



虽然在右边的图中代价函数逐渐减小，但是选择的学习速率$\eta=0.0001$的值太小，以致为了达到算法收敛的目标，需要更多的迭代次数。

下图说明了我们如何通过更改特定的权重参数值来最小化代价函数J（左子图）。右子图则展示了，如果学习速率选择过大会发生什么情况：算法跳过全局最优解（全局最小值）。



本书涉及的许多机器学习算法要求对特征值范围进行特征缩放，以优化算法的性能，我们将在第3章中做更深入的介绍。梯度下降就是通过特征缩放而受益的众多算法之一。在此，采用一种称作标准化的特征缩放方法，此方法可以使数据具备标准正态分布的特性：各特征值的均值为0，标准差为1。例如，为了对第j个特征的值进行标准化处理，只需要将其值与所有样本的平均值μ_j相减，并除以其标准差σ_j。

$$x_j' = \frac{x_j - \mu_j}{\sigma_j}$$

这里的x_j是包含训练样本n中第j个特征的所有值的向量。

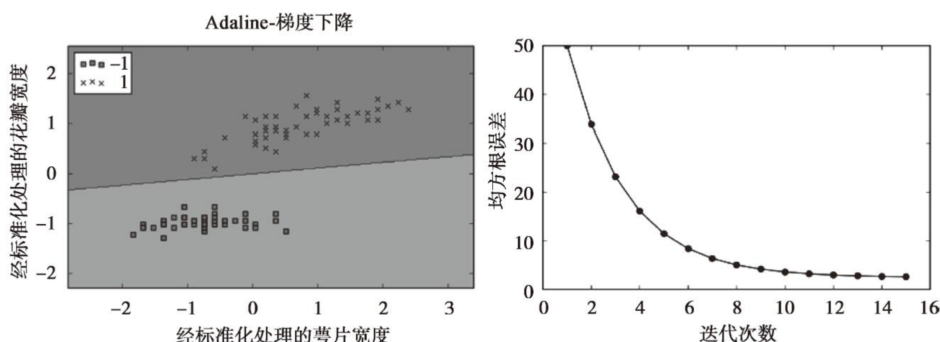
标准化可以简单地通过NumPy的mean和std方法来完成：

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

在进行标准化操作后，我们以学习速率η = 0.01再次对Adaline进行训练，看看它是否是收敛的：

```
>>> ada = AdalineGD(n_iter=15, eta=0.01)
>>> ada.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline - Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')
>>> plt.ylabel('petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Sum-squared-error')
>>> plt.show()
```

执行上述代码，我们可以看到图中判定区域，以及代价函数逐步减小的趋势，如下图所示：



如上图所示，以η = 0.01为学习速率，Adaline算法在经过标准化特征处理的数据上训练可以收敛。虽然所有样本都被正确分类，但是其误差平方和（SSE）的值仍旧不为零。

2.3.3　大规模机器学习与随机梯度下降

在上一节中，我们学习了如何使用整个训练数据集沿着梯度相反的方向进行优化，以最小化代价函数；这也是此方法有时称作批量梯度下降的原因。假定现在有一个包含几百万条数据的巨大数据集，对许多机器学习应用来说，这个量是非同寻常的。由于向全局最优点移动的每一步都需要使用整个数据集来进行评估，因此这种情况下使用批量梯度下降的计算成本非常高。

一个常用的替代批量梯度下降的优化算法是随机梯度下降（stochastic gradient descent），有时也称作迭代梯度下降（iterative gradient descent）或者在线梯度下降（on-line gradient descent）。与基于所有样本$x^{(i)}$的累积误差更新权重的策略不同：

$$\Delta \boldsymbol{w}=\eta\sum_i(y^{(i)}-\phi(z^{(i)}))\boldsymbol{x}^{(i)}$$

我们每次使用一个训练样本渐进地更新权重：

$$\eta(y^{(i)}-\phi(z^{(i)}))\boldsymbol{x}^{(i)}$$

虽然随机梯度下降可看作对梯度下降的近似，但是由于权重更新更频繁，通常它能更快收敛。由于梯度的计算是基于单个训练样本来完成的，因此其误差曲面不及梯度下降的平滑，但这也使得随机梯度下降更容易跳出小范围的局部最优点。为了通过随机梯度下降得到更加准确的结果，让数据以随机的方式提供给算法是非常重要的，这也是我们每次迭代都要打乱训练集以防止进入循环的原因。

当实现随机梯度下降时，通常使用随着时间变化的自适应学习速率来替代固定学习速率η，例如：$\frac{c_1}{[\text{迭代次数}]+c_2}$，其中$c_1$和$c_2$均为常数。

请注意，随机梯度下降不一定会得到全局最优解，但会趋近于它。借助于自适应学习速率，我们可以更进一步趋近于全局最优解。

随机梯度下降的另一个优势是我们可以将其用于在线学习。通过在线学习，当有新的数据输入时模型会被实时训练。这在我们面对海量数据时特别有效，例如针对Web中的用户信息。使用在线学习，系统可以及时地适应变化，同时如果考虑存储成本的话，也可以将训练数据在完成对模型的更新后丢弃。

小批次学习是介于梯度下降和随机梯度下降之间的一种技术。可以将小批次学习理解为在相对较小的训练数据子集上应用梯度下降——例如，一次使用50个样本。与梯度下降相比，由于权重的更新更加频繁，因此其收敛速度更快。此外，小批次学习使得我们可以用向量化操作来替代for循环，从而进一步提高学习算法的计算效率。

由于我们已经使用梯度下降实现了Adaline学习规则，因此只需在此基础上将学习算法中的权重更新改为通过随机梯度下降来实现即可。现在把fit方法改为使用单个训练样本来更新权重。此外，我们增加了一个partial_fit方法，对于在线学习，此方法不会重置权重。为了检验算法在训练后是否收敛，我们将每次迭代后计算出的代价值作为训练样本的平均消耗。此外，我们还增加了一个shuffle训练数据选项，每次迭代前重排训练数据避免在优化代价函数阶段陷入循环。通过random_state参数，我们可以指定随机数种子以保持多次训练的一致性。

```
from numpy.random import seed

class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclassifications in every epoch.
    shuffle : bool (default: True)
        Shuffles training data every epoch
        if True to prevent cycles.
    random_state : int (default: None)
        Set random state for shuffling
        and initializing the weights.

    """
    def __init__(self, eta=0.01, n_iter=10,
               shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        if random_state:
            seed(random_state)

    def fit(self, X, y):
```

```
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples
            is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
        self : object

        """
        self._initialize_weights(X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost)/len(y)
            self.cost_.append(avg_cost)
        return self

    def partial_fit(self, X, y):
        """Fit training data without reinitializing the weights"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Shuffle training data"""
        r = np.random.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Initialize weights to zeros"""
        self.w_ = np.zeros(1 + m)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Apply Adaline learning rule to update the weights"""
        output = self.net_input(xi)
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
```

```
        cost = 0.5 * error**2
        return cost

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Compute linear activation"""
        return self.net_input(X)

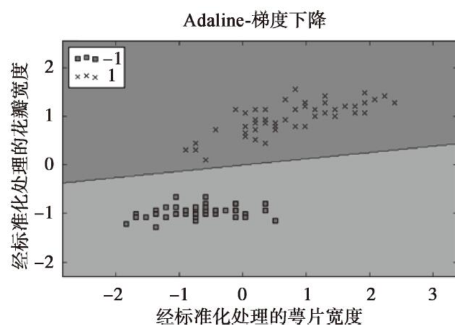
    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

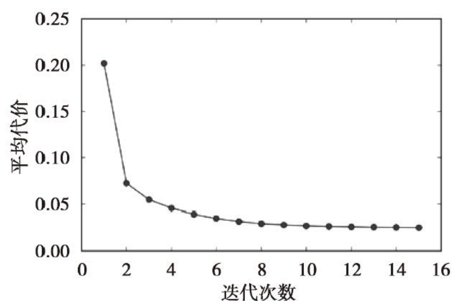
分类器AdalineSGD中_shuffle方法的工作原理如下：通过numpy.random的permutation函数，我们生成一个包含0～100的不重复的随机序列。这些数字可以作为索引帮助打乱我们的特征矩阵和类标向量。

接下来，我们就可以通过fit方法训练AdalineSGD分类器，并应用plot_decision_regions方法绘制训练结果：

```
>>> ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada)
>>> plt.title('Adaline - Stochastic Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')
>>> plt.ylabel('petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
>>> plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Average Cost')
>>> plt.show()
```

执行前面的示例代码，我们得到下图：





可以看到，代价函数的均值下降得很快，经过15次迭代后，最终的分类界面与使用梯度下降得到的Adaline分类界面类似。如果要改进模型，例如用于处理流数据的在线学习，可以对单个样本简单地调用parital_fit方法，如：ada.partial_fit（X_std[0,:],y[0]）。

本章小结

在本章中，我们仔细讲解了监督学习中线性分类器的基本概念。在实现了感知器算法后，我们学习了如何通过向量化的梯度下降高效地实现自适应线性神经元，以及通过随机梯度下降实现在线学习。至此，我们已经知道了如何使用Python来实现简单的分类器。在下一章中，我们将使用Python的scikit-learn机器学习库来获得一些现成的机器学习分类器，这些高效的分类器常应用于学术及工程领域。

第3章 使用scikit-learn实现机器学习分类算法

在本章中，我们从学术界和工业界常用的机器学习算法中，挑选一些流行高效的算法进行讲解。在了解监督学习分类算法之间差异的同时，我们也能对各个算法的优势及劣势能有个直观的认识。此外，我们还要使用scikit-learn库迈出第一步，它为高效使用机器学习算法提供了一个用户友好的接口。

在本章中，我们将学到如下内容：

·介绍常用分类算法的概念

·使用scikit-learn机器学习库

·选择机器学习算法时需要注意的问题

3.1　分类算法的选择

由于每个算法都基于某些特定的假设，且均含有某些缺点，因此需要通过大量的实践为特定的问题选择合适的算法。再次借用一下“没有免费午餐”理论：没有任何一种分类器可以在所有可能的应用场景下都有良好的表现。实践证明，只有比较了多种学习算法的性能，才能为特定问题挑选出最合适的模型。这些模型针对不同数量的特征或者样本、数据集中噪声的数量，以及类别是否线性可分等问题时，表现各不相同。

总而言之，分类器的性能、计算能力和预测能力，在很大程度上都依赖于用于模型训练的相关数据。训练机器学习算法所涉及的五个主要步骤可以概述如下：

1.特征的选择

2.确定性能评价标准

3.选择分类器及其优化算法

4.对模型性能的评估

5.算法的调优

由于本书通过循序渐进的方式讲解机器学习的相关内容，因此本章将重点介绍不同分类算法的基本概念，并再次回顾特征选择、预处理及性能评价标准等内容，而关于超参调优等更详细的内容将在本书的后续章节讨论。

3.2　初涉scikit-learn的使用

在第2章中，我们学习了两种相关的分类算法：感知器规则算法和Adaline算法，我们使用Python自行实现了这两种算法。现在，了解一下scikit-learn的API，它在实现几种高度优化的分类算法的同时，还提供了一个用户友好的接口。不过，scikit-learn库不仅提供了大量的学习算法，还包含许多用于对数据进行预处理、调优和对模型评估的功能。第4章和第5章会讨论这些问题和相关概念。

使用scikit-learn训练感知器

首先，我们使用scikit-learn训练一个感知器模型，此模型与第2章中实现的感知器模型类似。为了简单起见，我们将在本章后续几节中使用我们已经熟悉的鸢尾花数据集。由于鸢尾花数据集是一个简单、流行的数据集，它经常用于算法实验与测试，因此它已经默认包含在scikit-learn库中。同样，出于可视化应用的考虑，我们仍旧只使用鸢尾花数据集中的两个特征。

提取150个花朵样本中的花瓣长度和花瓣宽度两个特征的值，并由此构建特征矩阵X，同时将对应花朵所属类型的类标赋值给向量y：

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
```

如果执行np.unique（y）返回存储在iris.target中的各类花朵的类标，

可以看到：scikit-learn已分别将Iris-Sentosa、Iris-Versicolor和Iris-Virginia的类名另存为整数（0,1,2），对许多机器学习库来说，这是针对性能优化一种推荐的做法。

为了评估训练得到的模型在未知数据上的表现，我们进一步将数据集划分为训练数据集和测试数据集。第5章会更加详细地讨论关于模型验证的操作细节。

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...             X, y, test_size=0.3, random_state=0)
```

使用scikit-learn中cross_validation模块中的train_test_split函数，随机将数据矩阵X与类标向量y按照3：7的比例划分为测试数据集（45个样本）和训练数据集（105个样本）。

第2章关于梯度下降的例子已经提到：为了优化性能，许多机器学习和优化算法都要求对数据做特征缩放。在此，我们将使用scikit-learn的preprocessing模块中的StandardScaler类对特征进行标准化处理。

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

在上面的代码中，从preprocessing模块中加载了StandardScaler类，并实例化了一个StandardScaler对象，用变量sc作为对它的引用。使用StandardScaler中的fit方法，可以计算训练数据中每个特征的μ（样本均值）和σ（标准差）。通过调用transform方法，可以使用前面计算得到的μ和σ来对训练数据做标准化处理。需注意的是，我们要使用相同的缩放参

数分别处理训练和测试数据集，以保证它们的值是彼此相当的。

在对训练数据做了标准化处理后，我们现在可以训练感知器模型了。scikit-learn中的大多数算法本身就使用一对多（One-vs-Rest，OvR）方法来支持多类别分类，因此，我们可以将三种鸢尾花的数据同时输入到感知器中。代码如下：

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)
>>> ppn.fit(X_train_std, y_train)
```

对于使用scikit-learn的接口训练感知器，其流程与我们在第2章中实现的感知器的流程类似：在加载了linear_model模块中的Perceptron类后，我们实例化了一个新的Perceptron对象，并通过fit方法训练模型。此模型中的参数eta0与我们自行实现的感知器中的学习速率eta等价，而参数n_iter定义了迭代的次数（遍历训练数据集的次数）。第2章提到：合适的学习速率需要通过实验来获取。如果学习速率太大，算法可能会跳过全局最优点；如果学习速率太小，算法将需要更多次的迭代以达到收敛，这将导致训练速度变慢——尤其是面临巨大的数据集时。此外，我们使用random_state参数在每次迭代后初始化重排训练数据集。

与第2章中实现的感知器一样，使用scikit-learn完成模型的训练后，就可以在测试数据集上使用predict方法进行预测了，代码如下：

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified samples: %d' % (y_test != y_pred).sum())
Misclassified samples: 4
```

执行上述代码后，可以看到，在感知器对45朵花的分类结果中，有4个是错误的。也就是在测试数据集上的错误率为0.089或者说是

8.9%（4/45约等于0.089）。

许多机器学习从业者通常使用准确率而不是误分类率来评判一个模型，它们之间的关系如下：

1-误分类率 = 0.911或91.1%

在metrics模块中，scikit-learn还实现了许多不同的性能矩阵。例如，可以通过下列代码计算感知器在测试数据集上的分类准确率：

```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
0.91
```

此处，y_test是真实的类标，而y_pred是通过前面预测得到的类标。

本章我们使用测试集评估模型的性能。在第5章中，读者将学到使用诸如学习曲线等图形分析工具来检测并预防过拟合（overfitting）。过拟合意味着模型很好地捕获到了训练数据集中的模式，但是却无法泛化到未知的新数据上。

最后，可以使用在第2章中实现的plot_decision_regions函数来绘制刚刚训练过的模型的决策区域，并观察不同花朵样本的分类效果。不过，我们做少许修改：使用小圆圈来高亮显示来自测试数据集的样本：

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

def plot_decision_regions(X, y, classifier,
                    test_idx=None, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

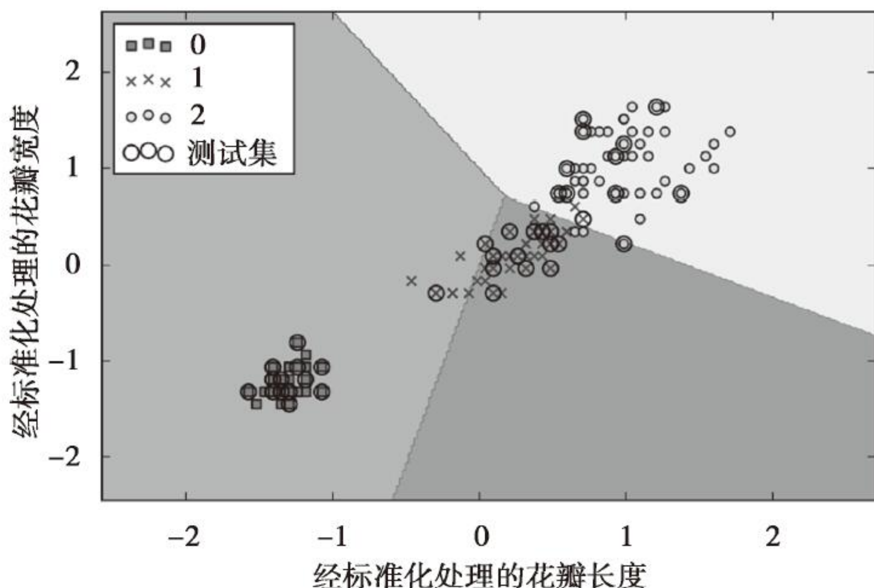
    # plot all samples
    X_test, y_test = X[test_idx, :], y[test_idx]
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)

    # highlight test samples
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1], c='',
                alpha=1.0, linewidth=1, marker='o',
                s=55, label='test set')
```

对plot_decision_regions函数稍做修改后（前面代码中加重显示的部分），可以在结果呈现的图像中区分出哪些是我们需要预测的样本。代码如下：

```
>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                       y=y_combined,
...                       classifier=ppn,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

从结果呈现的图像中我们可以看到，无法通过一个线性决策边界完美区分三类样本。



回忆一下第2章讨论过的内容：对于无法完美线性可分的数据集，感知器算法将永远无法收敛，这也是在实践中一般不推荐使用感知器算法的原因。后续几节，我们将着眼于更加先进的线性分类器算法，即使是面对无法完美线性可分的数据集，这些算法也可以在一定程度上收敛。

Perceptron以及scikit-learn中其他的一些函数和类都包含许多额外的参数，为了清晰起见，我们忽略了这些参数。读者可以通过Python中的help函数（如help（Perceptron）），或者通过scikit-learn的在线文档（http://scikit-learn.org/stable/）来了解这些参数的相关信息。

3.3 逻辑斯谛回归中的类别概率

感知器是机器学习分类算法中优雅易用的一个入门级算法，不过其最大的缺点在于：在样本不是完全线性可分的情况下，它永远不会收敛。上一小节中关于分类的任务就是这样的一个例子。直观上，可以把原因归咎于：在每次迭代过程中，总是存在至少一个分类错误的样本，从而导致了权重持续更新。当然，你也可以改变学习速率并增加迭代次数，不过感知器在此类数据集上仍旧永远无法收敛。为了提高分类的效率，我们学习另外一种针对线性二类别分类问题的简单但更高效的算法：逻辑斯谛回归（logistic regression）。请注意：不要被其名字所迷惑，逻辑斯谛回归是一个分类模型，而不是回归模型。

3.3.1　初识逻辑斯谛回归与条件概率

逻辑斯谛回归是针对线性可分问题的一种易于实现且性能优异的分类模型。它是业界应用最为广泛的分类模型之一。与感知器及Adaline类似，逻辑斯谛回归模型也是适用于二类别分类问题的线性模型，通过一对多（OvR）技术可以扩展到多类别分类。

在介绍逻辑斯谛回归作为一种概率模型所具有的特性之前，我们先介绍一下几率比（odd ratio）[1]，它指的是特定事件发生的几率。用数学公式表示为$\frac{p}{(1-p)}$，其中p为正事件发生的概率。此处，正事件并不意味着好的事件，而是指我们所要预测的事件，以一个患者患有某种疾病的概率为例，我们可以将正事件的类标标记为y = 1。更进一步，我们可以定义logit函数，它是几率比的对数函数（log-odds，对数几率）。

$$\text{logit}(p)=\log\frac{p}{(1-p)}$$

logit函数的输入值范围介于区间[0,1]，它能将输入转换到整个实数范围内，由此可以将对数几率记为输入特征值的线性表达式：

$$\text{logit}(p(y=1\mid x))=w_0x_0+w_1x_1+\cdots+w_mx_m=\sum_{i=0}^{n}w_mx_m=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$$

此处，p（y = 1|x）是在给定特征x的条件下，某一个样本属于类别1的条件概率。

我们在此的真正目的是预测某一样本属于特定类别的概率，它是logit函数的反函数，也称作logistic函数，由于它的图像呈S形，因此有时也简

称为sigmoid函数：

$$\phi(z)=\frac{1}{1+e^{-z}}$$

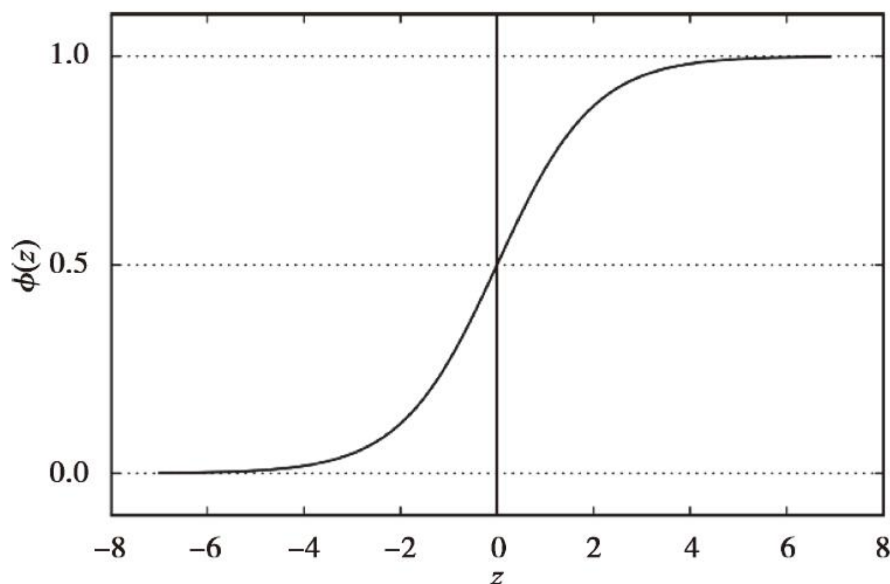
这里的z为净输入，也就是样本特征与权重的线性组合，其计算方式为：

$$z=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}=w_0x_0+w_1x_1+\cdots+w_mx_m$$

我们来绘制一下自变量取值介于区间[-7,7]的sigmoid函数的图像：

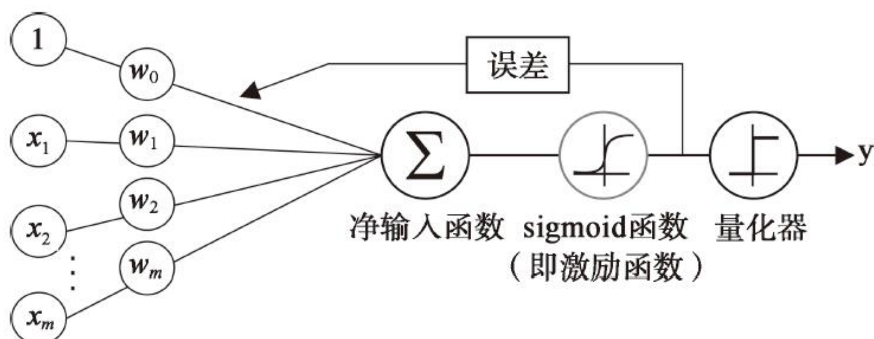
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> phi_z = sigmoid(z)
>>> plt.plot(z, phi_z)
>>> plt.axvline(0.0, color='k')
>>> plt.axhspan(0.0, 1.0, facecolor='1.0', alpha=1.0, ls='dotted')
>>> plt.axhline(y=0.5, ls='dotted', color='k')
>>> plt.yticks([0.0, 0.5, 1.0])
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi (z)$')
>>> plt.show()
```

执行上述代码，将会呈现一个S形（sigmoidal）曲线：



可以看到，当z趋向于无穷大（z→∞）时，φ（z）趋近于1，这是由于e^{-z}在z值极大的情况下其值变得极小。类似地，当z趋向于负无穷（z→-∞）时，φ（z）趋近于0，这是由于此时分母越来越大的结果。由此，可以得出结论：sigmoid函数以实数值作为输入并将其映射到[0,1]区间，其拐点位于φ（z）=0.5处。

为了对逻辑斯谛回归模型有个直观的认识，我们可以将其与第2章介绍的Adaline联系起来。在Adaline中，我们使用恒等函数φ（z）=z作为激励函数。而在逻辑斯谛回归中，只是简单地将前面提及的sigmoid函数作为激励函数，如下图所示：



在给定特征x及其权重w的情况下，sigmoid函数的输出给出了特定样本x属于类别1的概率φ（z）＝P（y＝1｜x;w）。例如，针对某一样本我们求得φ（z）＝0.8，这意味着该样本属于变色鸢尾的概率为80%。类似地，该样本属于山鸢尾的概率可计算为P（y＝0｜x;w）＝1-P（y＝1｜x;w）＝0.2，也就是20%。预测得到的概率可以通过一个量化器（单位阶跃函数）简单地转换为二元输出：

$$\hat{y}=\begin{cases}1 & 若\phi(z)\geqslant 0.5\\ 0 & 其他\end{cases}$$

对照前面给出的sigmoid函数的图像，它其实相当于：

$$\hat{y}=\begin{cases}1 & 若\ z\geqslant 0.0\\ 0 & 其他\end{cases}$$

对于许多应用实践来说，我们不但对类标预测感兴趣，而且对事件属于某一类别的概率进行预测也非常有用。例如，将逻辑斯谛回归应用于天气预报，不仅要预测某天是否会下雨，还要推测出下雨有多大的可能性。同样，逻辑斯谛回归还可用于预测在出现某些症状的情况下，患者患有某种疾病的可能性，这也是逻辑斯谛回归在医疗领域得到广泛应用的原因。

[1] 某一事件发生与不发生的概率的比值。——译者注

3.3.2 通过逻辑斯谛回归模型的代价函数获得权重

我们已经学会了如何使用逻辑斯谛回归模型预测概率和类标。现在简要介绍一下模型的参数，例如这里的权重w。在上一章中，我们将其代价函数定义为误差平方和（sum-squared-error）：

$$J(\boldsymbol{w})=\sum_i \frac{1}{2}\left(\phi\left(z^{(i)}\right)-y^{(i)}\right)^2$$

通过最小化此代价函数，我们可以得到Adaline分类模型的权重w。为了推导出逻辑斯谛回归模型的代价函数，我们在构建逻辑斯谛回归模型时，需要先定义一个最大似然函数L，假定数据集中的每个样本都是相互独立的，其计算公式如下：

$$L(\boldsymbol{w})=P(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{w})=\prod_{i=1}^{n} P\left(y^{(i)}|x^{(i)};\boldsymbol{w}\right)=\left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}}\left(1-\phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

在实际应用中，很容易对此方程的（自然）对数进行最大化处理（求其极大值），故定义了对数似然函数：

$$l(\boldsymbol{w})=\log L(\boldsymbol{w})=\sum_{i=1}^{n} \log\left(\phi\left(z^{(i)}\right)\right)+\left(1-y^{(i)}\right)\log\left(1-\phi\left(z^{(i)}\right)\right)$$

首先，在似然函数值非常小的时候，可能出现数值溢出的情况，使用对数函数降低了这种情况发生的可能性。其次，我们可以将各因子的连乘转换为和的形式，利用微积分中的方法，通过加法转换技巧可以更容易地对函数求导。

现在可以通过梯度上升等优化算法最大化似然函数。或者，改写一下

对数似然函数作为代价函数J，这样就可以使用第2章中的梯度下降做最小化处理了。

$$J(\boldsymbol{w})=\sum_{i=1}^{n}-\log(\phi(z^{(i)}))-(1-y^{(i)})\log(1-\phi(z^{(i)}))$$

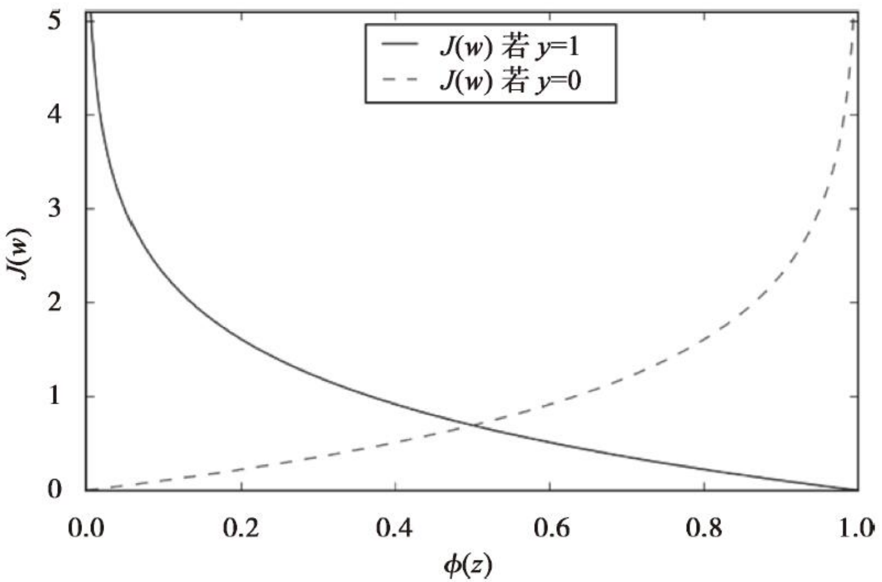
为了更好地理解这一代价函数，先了解一下如何计算单个样本实例的成本：

$$J(\phi(z),y;\boldsymbol{w})=-y\log(\phi(z))-(1-y)\log(1-\phi(z))$$

通过上述公式可以发现：如果y = 0，则第一项为0；如果y = 1，则第二项为0：

$$J(\phi(z),y;\boldsymbol{w})=\begin{cases}-\log(\phi(z)) & 若\ y=1\\ -\log(1-\phi(z)) & 若\ y=0\end{cases}$$

下图解释了在不同φ（z）值时对单一示例样本进行分类的代价：



可以看到，如果将样本正确划分到类别1中，代价将趋近于0（实线）。类似地，如果正确将样本划分到类别0中，y轴所代表的代价也将趋近于0（虚线）。然而，如果分类错误，代价将趋近于无穷。这也就意味着错误预测带来的代价将越来越大。

3.3.3 使用scikit-learn训练逻辑斯谛回归模型

如果想要自己编写代码实现逻辑斯谛回归，只要将第2章中的代价函数J替换为下面的代价函数：

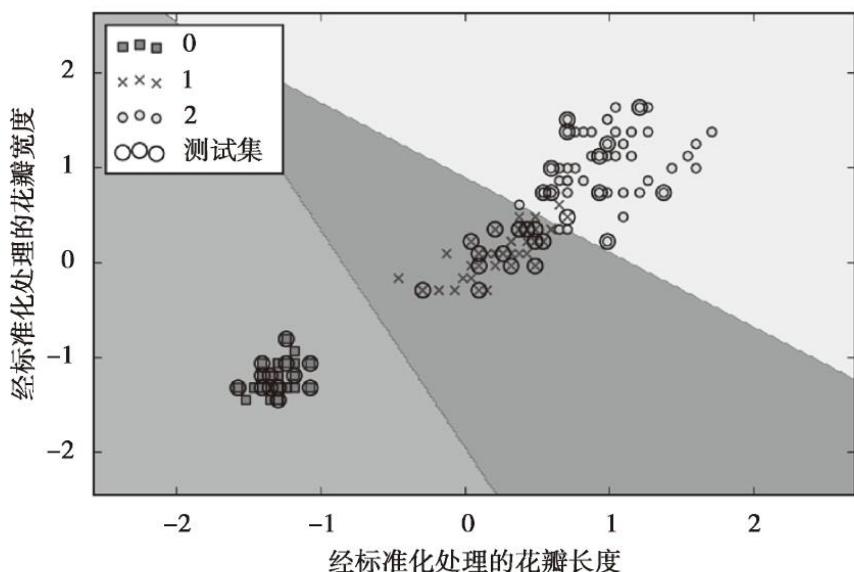
$$J(\boldsymbol{w})=-\sum_{i} y^{(i)}\log(\phi(z^{(i)}))+(1-y^{(i)})\log(1-\phi(z^{(i)}))$$

该函数通过计算每次迭代过程中训练所有样本的成本，最终得到一个可用的逻辑斯谛回归模型。然而，由于scikit-learn已经有现成的经过高度优化的逻辑斯谛算法，而且还支持多类别分类，因此，我们不再尝试自己实现算法，转而使用sklearn.linear_model.LogisticRegression类来完成，同时使用我们已经非常熟悉的fit方法在鸢尾花训练数据集上训练模型。

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=1000.0, random_state=0)

>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=lr,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

在训练集上对模型进行训练后，我们在二维图像中绘制了决策区域、训练样本和测试样本，如下图所示：



分析前面用来训练LogisticRegression模型的代码，你也许会思考："那个神秘的参数C代表什么？"下一节会先介绍一下过拟合和正则化的概念，然后再来讨论这个神秘的参数。

此外，可以通过predict_proba方法来预测样本属于某一类别的概率。例如，可以预测第一个样本属于各个类别的概率：

```
>>> lr.predict_proba(X_test_std[0,:])
```

通过此代码可得到如下数组：

```
array([[  0.000,   0.063,   0.937]])
```

此数组表明模型预测此样本属于Iris-Virginica的概率为93.7%，属于Iris-Versicolor的概率为6.3%。

可以证明，逻辑斯谛回归中的回归系数更新用到的梯度下降本质上与

第2章中的公式是相同的。首先，计算对数似然函数对第j个权重的偏导：

$$\frac{\partial}{\partial w_j} l(\boldsymbol{w}) = \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

在进入下一步之前，先计算一下sigmoid函数的偏导：

$$\frac{\partial}{\partial w_j} \phi(z) = \frac{\partial}{\partial z} \frac{1}{1+e^{-z}} = \frac{1}{(1+e^{-z})^2} e^{-z} = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}} \right) = \phi(z)(1-\phi(z)$$

将$\frac{\partial}{\partial w_j} \phi(z) = \phi(z)(1-\phi(z))$代入第一个方程中，可以得到：

$$\begin{aligned}
&\left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z) \\
&= \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \phi(z)(1-\phi(z)) \frac{\partial}{\partial w_j} z \\
&= (y(1-\phi(z)) - (1-y)\phi(z)) x_j \\
&= (y - \phi(z)) x_j
\end{aligned}$$

我们的目标是求得能够使对数似然函数最大化的权重值，在此需按如下公式更新所有权重：

$$w_j := w_j + \eta \sum_{i=1}^{n} (y^{(i)} - \phi(z^{(i)})) x^{(i)}$$

由于我们是同时更新所有的权重的，因此可以将更新规则记为：

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$$

其中，Δw定义为：

$$\Delta \boldsymbol{w} = \eta \nabla l(\boldsymbol{w})$$

由于最大化对数似然函数等价于最小化前面定义的代价函数J，因此可以将梯度下降的更新规则定义为：

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^{n} (y^{(i)} - \phi (z^{(i)}) x^{(i)})$$

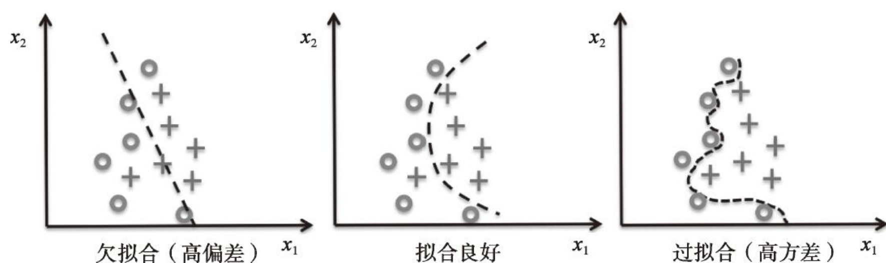
$$\boldsymbol{w} := \boldsymbol{w} + \Delta w, \ \Delta w = -\eta \nabla J(\boldsymbol{w})$$

这等价于第2章中的梯度下降规则。

3.3.4 通过正则化解决过拟合问题

过拟合是机器学习中的常见问题，它是指模型在训练数据集上表现良好，但是用于未知数据（测试数据）时性能不佳。如果一个模型出现了过拟合问题，我们也说此模型有高方差，这有可能是因为使用了相关数据中过多的参数，从而使得模型变得过于复杂。同样，模型也可能面临欠拟合（高偏差）问题，这意味着模型过于简单，无法发现训练数据集中隐含的模式，这也会使得模型应用于未知数据时性能不佳。

虽然我们目前只介绍了分类中的线性模型，但对于过拟合与欠拟合问题，最好使用更加复杂的非线性决策边界来阐明，如下图所示：



如果我们多次重复训练一个模型，如使用训练数据集中不同的子集，方差可以用来衡量模型对特定样本实例预测的一致性（或者说变化）。可以说模型对训练数据中的随机性是敏感的。相反，当我们在不同的训练数据集上多次重建模型时，偏差可以从总体上衡量预测值与实际值之间的差异；偏差并不是由样本的随机性导致的，它衡量的是系统误差。

偏差-方差权衡（bias-variance tradeoff）就是通过正则化调整模型的

复杂度。正则化是解决共线性（特征间高度相关）的一个很有用的方法，它可以过滤掉滤数据中的噪声，并最终防止过拟合。正则化背后的概念是引入额外的信息（偏差）来对极端参数权重做出惩罚。最常用的正则化形式称为L2正则化（L2 regularization），它有时也称作L2收缩（L2 shrinkage）或权重衰减（weight decay），可写作：

$$\frac{\lambda}{2}\|\boldsymbol{w}\|^2=\frac{\lambda}{2}\sum_{j=1}^{m}w_j^2$$

其中，λ为正则化系数。

特征缩放（如标准化等）之所以重要，其中一个原因就是正则化。为了使得正则化起作用，需要确保所有特征的衡量标准保持统一。

使用正则化方法时λ，我们只需在逻辑斯谛回归的代价函数中加入正则化项，以降低回归系数带来的副作用：

$$J(\boldsymbol{w})=\left\{\sum_{i=1}^{n}(-\log(\phi(z^{(i)}))+(1-y^{(i)}))(-\log(1-\phi(z^{(i)})))\right\}+\frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

通过正则化系数，保持权值较小时，我们就可以控制模型与训练数据的拟合程度。增加λ的值，可以增强正则化的强度。

前面用到了scikit-learn库中的LogisticRegression类，其中的参数C来自下一小节要介绍的支持向量机中的约定，它是正则化系数的倒数：

$$C=\frac{1}{\lambda}$$

由此，我们可以将逻辑斯谛回归中经过正则化的代价函数写作：

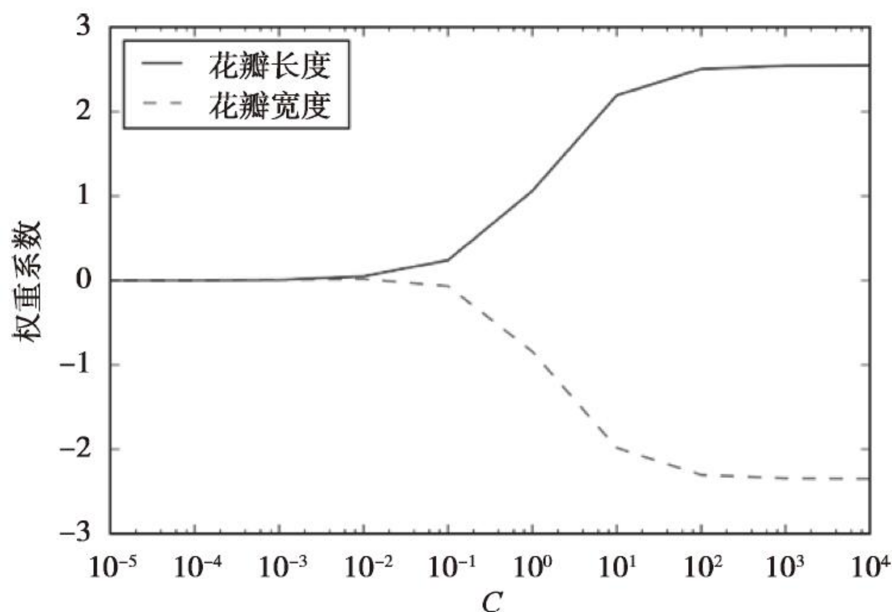
$$J(w)=C\left\{\sum_{i=1}^{n}(-\log(\phi(z^{(i)}))+(1-y^{(i)}))(-\log(1-\phi(z^{(i)})))\right\}+\frac{1}{2}||w||^2$$

因此，减小正则化参数倒数C的值相当于增加正则化的强度，这可以通过绘制对两个权重系数进行L2正则化后的图像予以展示：

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10**c, random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...          label='petal length')
>>> plt.plot(params, weights[:, 1], linestyle='--',
...          label='petal width')
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

执行上述代码，我们使用不同的逆正则化参数C拟合了10个逻辑斯谛回归模型。出于演示的目的，我们仅仅记录了类别2区别于其他类别的权重系数。请牢记，我们使用OvR技术来实现多类别分类。

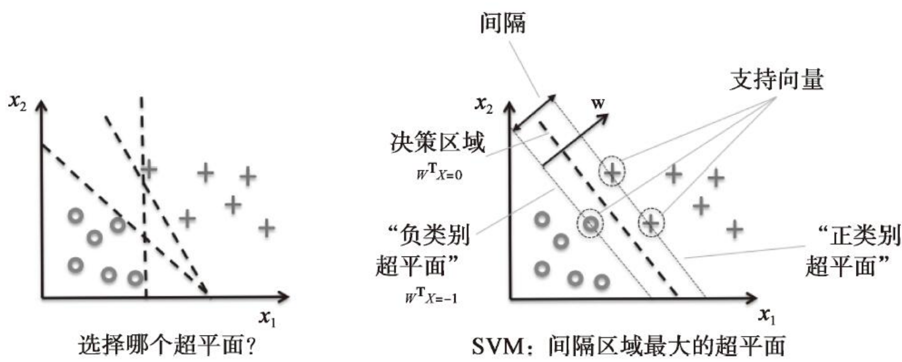
通过结果图像可以看到，如果我们减小参数C的值，也就是增加正则化项的强度，可以导致权重系数逐渐收缩。



由于深入讲解每个分类算法的细节已经超出了本书的讨论范围，因此强烈建议读者阅读Scott Menard博士的书籍（Logistic Regression:From Introductory to Advanced Concepts and Applications,Sage Publications）以了解更多关于逻辑斯谛回归的内容。

3.4 使用支持向量机最大化分类间隔

另一种性能强大且广泛应用的学习算法是支持向量机（support vector machine，SVM），它可以看作对感知器的扩展。在感知器算法中，我们可以最小化分类误差。而在SVM中，我们的优化目标是最大化分类间隔。此处间隔是指两个分离的超平面（决策边界）间的距离，而最靠近超平面的训练样本称作支持向量（support vector），如下图所示：



3.4.1 对分类间隔最大化的直观认识

决策边界间具有较大的间隔意味着模型具有较小的泛化误差，而较小的间隔则意味着模型可能过拟合。为了对间隔最大化有个直观的认识，我们仔细观察一下两条平行的决策边界，我们分别称其为正、负超平面，可表示为：

$$w_0 + \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_{pos} = 1 \qquad (1)$$

$$w_0 + \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_{neg} = -1 \qquad (2)$$

如果我们将等式（1）（2）相减，可以得到：

$$\Rightarrow \boldsymbol{w}^{\mathrm{T}} (\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg}) = 2$$

我们可以通过向量w的长度来对其进行规范化，做如下定义：

$$||\boldsymbol{w}|| = \sqrt{\sum_{j=1}^{m} w_j^2}$$

由此可以得到如下等式：

$$\frac{\boldsymbol{w}^{\mathrm{T}} (\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg})}{||\boldsymbol{w}||} = \frac{2}{||\boldsymbol{w}||}$$

上述等式的左侧可以解释为正、负超平面间的距离，也就是我们要最大化的间隔。

在样本正确划分的前提下，最大化分类间隔也就是使$\frac{2}{||w||}$最大化，这也是SVM的目标函数，记为：

$$w_0 + \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^{(i)} \geqslant 1 \text{若} y^{(i)} = 1$$

$$w_0 + \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^{(i)} < -1 \text{若} y^{(i)} = -1$$

这两个方程可以解释为：所有的负样本都落在负超平面一侧，而所有的正样本则在正超平面划分出的区域中。它们可以写成更紧凑的形式：

$$y^{(i)}(w_0 + \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^{(i)}) \geqslant 1 \forall_i$$

在实践中，通过二次规划的方法，很容易对目标函数的倒数项$\frac{1}{2}\|w\|$进行最小化处理。不过关于二次规化的详细内容超出了本书的范围，如果读者感兴趣，可以通过阅读由Springer出版社出版，Vladimir Vapnik的The Nature of Statistical Learning Theory一书，或者查阅Chris J.C.Burger在其论文“A Tutorial on Support Vector Machines for Pattern Recognition”（发表于Data mining and knowledge discovery，2（2）:121～167，1998）中的精彩解释。

3.4.2 使用松弛变量解决非线性可分问题

我们无需对间隔分类背后的数学概念进行更深入的探讨，在此只是简单介绍一下松弛变量ξ的概念。它由Vladimir Vapnik在1995年引入，并由此提出所谓的软间隔分类。介绍松弛系数ξ的目的是：对于非线性可分的数据来说，需要放松线性约束条件，以保证在适当的罚项成本下，对错误分类的情况进行优化时能够收敛。

取值为正的松弛变量可以简单地加入到线性约束中：

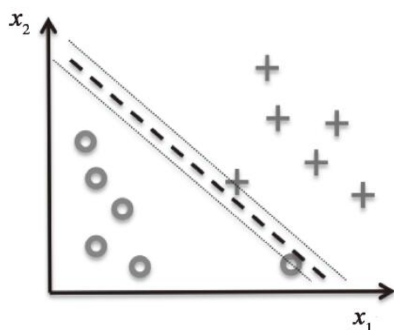
$$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^{(i)} \geqslant 1 \text{若} y^{(i)} = 1 - \xi^{(i)}$$

$$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^{(i)} < -1 \text{若} y^{(i)} = 1 + \xi^{(i)}$$

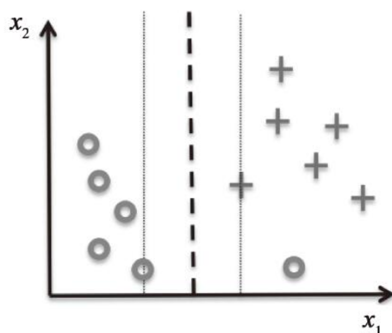
由此，在前面提及的约束条件下，新的优化目标为最小化下式：

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

通过变量C，我们可以控制对错误分类的惩罚程度。C值较大时，对应大的错误惩罚，当选择小的C值时，则对错误分类的惩罚没那么严格。因此，可以使用参数C来控制间隔的大小，进而可以在偏差和方差之间取得平衡，如下图所示：



当参数C的值较大时



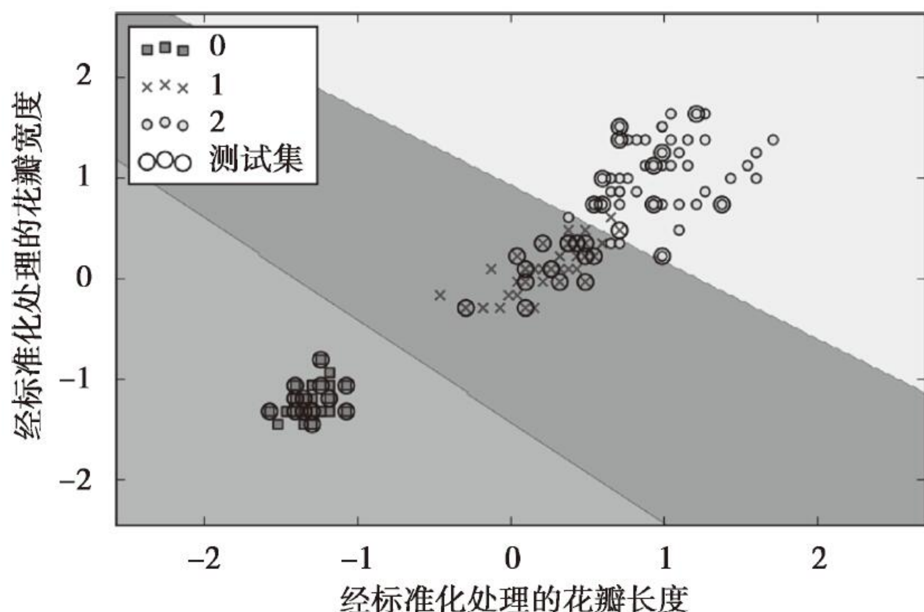
当参数C的值较小时

前面在回归的正则化中已经做了讨论，我们通过增加C的值来增加偏差并降低模型的方差。

至此，我们已经学习了线性SVM的基本概念，现在让我们来训练一个SVM模型以对鸢尾花数据集中的样本进行分类。

```
>>> from sklearn.svm import SVC
   >>> svm = SVC(kernel='linear', C=1.0, random_state=0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

执行上述代码可以对通过SVM获得的决策区域进行可视化，结果如下图所示：



逻辑斯谛回归与支持向量机

在实际分类任务中，线性逻辑斯谛回归与支持向量机往往能得到非常相似的结果。逻辑斯谛回归会尽量最大化训练数据集的条件似然，这使得它比支持向量机更易于处理离群点。而支持向量机则更关注接近决策边界（支持向量）的点。逻辑斯谛回归的另一个优势在于模型简单从而更容易实现。此外，逻辑斯谛回归模型更新方便，当应用于流数据分析时，这是非常具备吸引力的。

3.4.3　使用scikit-learn实现SVM

前面章节中用到了scikit-learn中的Perceptron和LogisticRegression类，它们都使用了LIBLINEAR库，LIBLINEAR是一个由中国台湾大学（http://www.csie.ntu.edu.tw/~cjlin/liblinear/）使用C/C++语言开发的经过高度优化的库。类似地，用于训练SVM模型的SVC类使用了LIBSVM库，它是一个专门用于SVM的C/C++库（http://www.csie.ntu.edu.tw/~cjlin/libsvm/）。

与使用原生Python代码实现线性分类器相比，使用LIBLINEAR和LIBSVM库实现有其优点：在用于大型数据时，可以获得极高的训练速度。然而，有些时候数据集非常巨大以致无法加载到内存中，针对这种情况，scikit-learn提供了SGDClassifier类供用户选择，这个类还通过partial_fit方法支持在线学习。SGDClassifier类背后的概念类似于第2章中实现的随机梯度算法。我们可以使用默认参数以如下方式分别初始化基于随机梯度下降的感知器、逻辑斯谛回归以及支持向量机模型。

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
>>> svm = SGDClassifier(loss='hinge')
```

3.5 使用核SVM解决非线性问题

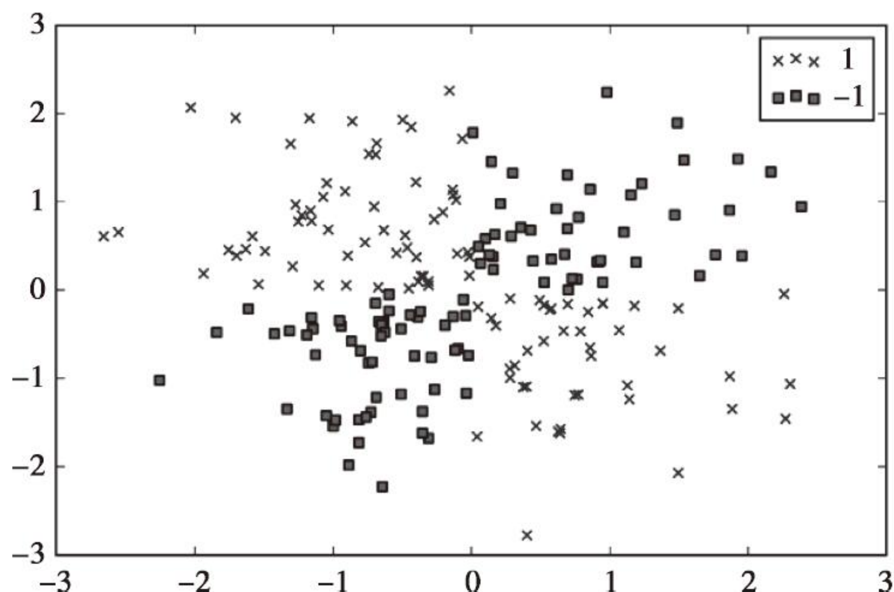
支持向量机在机器学习爱好者中广受欢迎的另一原因是：它可以很容易地使用“核技巧”来解决非线性可分问题。在讨论SVM的核的基本概念之前，先通过一个例子来认识一下所谓的非线性可分问题到底是什么。

通过如下代码，我们使用NumPy中的logical_xor函数创建了一个经过“异或”操作的数据集，其中100个样本属于类别1，另外的100个样本被划定为类别-1：

```
>>> np.random.seed(0)
>>> X_xor = np.random.randn(200, 2)
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0, X_xor[:, 1] > 0)
>>> y_xor = np.where(y_xor, 1, -1)

>>> plt.scatter(X_xor[y_xor==1, 0], X_xor[y_xor==1, 1],
...             c='b', marker='x', label='1')
>>> plt.scatter(X_xor[y_xor==-1, 0], X_xor[y_xor==-1, 1],
...             c='r', marker='s', label='-1')
>>> plt.ylim(-3.0)
>>> plt.legend()
>>> plt.show()
```

在执行上述代码后，我们通过随机噪声得到一个“异或”数据集，其二维分布图像如下所示：

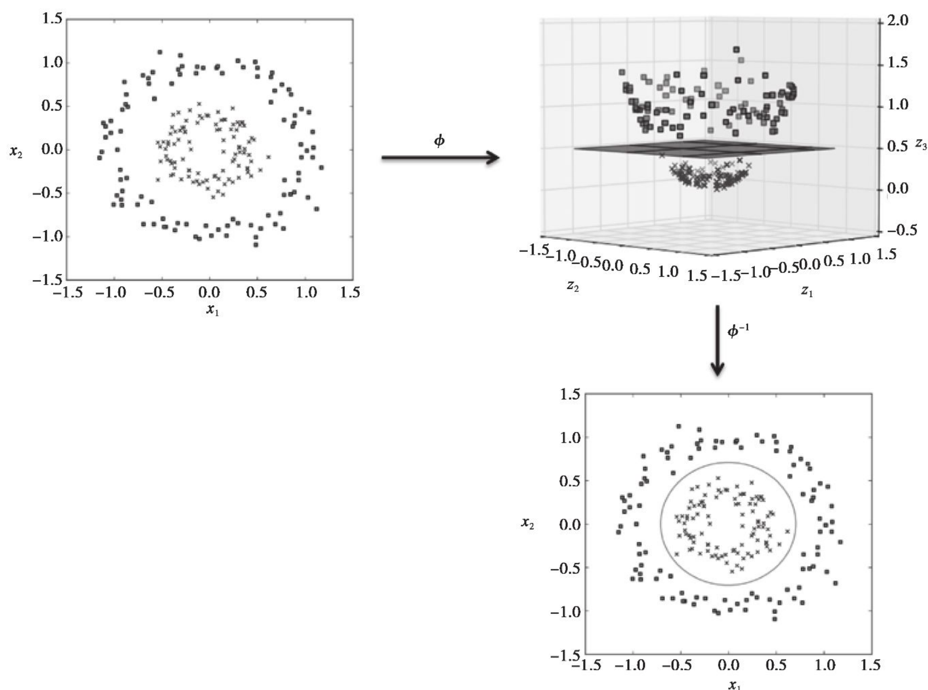


显然，如果使用前面小节讨论过的线性逻辑斯谛回归或者线性SVM模型，并将线性超平面当做决策边界，无法将样本正确地划分为正类别或负类别。

核方法处理此类非线性可分数据的基本理念就是：通过映射函数φ（·）将样本的原始特征映射到一个使样本线性可分的更高维空间中。如下图所示，我们可以将二维数据集通过下列映射转换到新的三维特征空间中，从而使得样本可分：

$$\phi(x_1, x_2)=(z_1, z_2, z_3)=(x_1, x_2, x_1^2+x_2^2)$$

这使得我们可以将图中的两个类别通过线性超平面进行分割，如果我们把此超平面映射回原始特征空间，则可线性分割两类数据的超平面就变为非线性的了。



使用核技巧在高维空间中发现分离超平面

为了使用SVM解决非线性问题，我们通过一个映射函数φ（·）将训练数据映射到更高维的特征空间，并在新的特征空间上训练一个线性SVM模型。然后将同样的映射函数φ（·）应用于新的、未知数据上（即使用此映射将未知数据映射到新的特征空间），进而使用新特征空间上的线性SVM模型对其进行分类。

但是，这种映射方法面临的一个问题就是：构建新的特征空间带来非常大的计算成本，特别是在处理高维数据的时候。这时就用到了我们称作核技巧的方法。我们不会过多关注SVM训练中所需解决的二次规划问题，在实践中，我们所需做的就是将点积$x^{(i)T}x^{(j)}$映射为φ（$x^{(i)}$）

$^T\varphi(x^{(j)})$。为了降低两点之间内积精确计算阶段的成本耗费，我们定义一个所谓的核函数：$k(x^{(i)}, x^{(j)}) = \varphi(x^{(i)})^T\varphi(x^{(j)})$。

一个最广为使用的核函数就是径向基函数核（Radial Basis Function kernel，RBF kernel）或高斯核（Gaussian kernel）：

$$k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \exp\left(-\frac{\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|^2}{2\sigma^2}\right)$$

此公式常简写为：

$$k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \exp(-\gamma\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|^2)$$

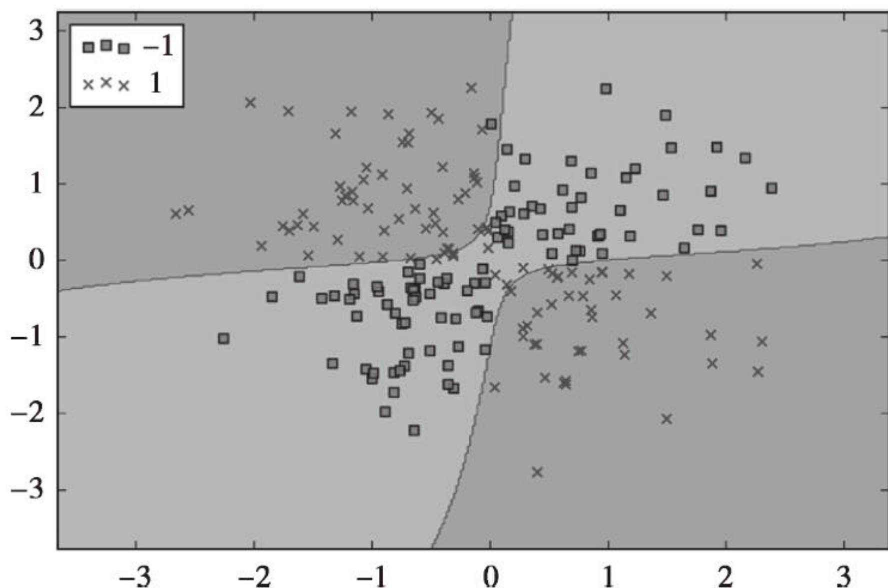
这里，$\gamma = \frac{1}{2\sigma^2}$是待优化的自由参数。

粗略地说，“核”可解释为一对样本之间的“相似函数”。此处的负号将距离转换为相似性评分，而由于指数项的存在，使得相似性评分会介于0之间（差异巨大的样本）和1（完全相同的样本）。

现在我们已经知道了使用核技巧的重点，尝试能否训练一个核SVM，使之可以通过一个非线性决策边界来对“异或”数据进行分类。在此，我们只是简单使用前面已经导入的scikit-learn包中的SVM类，并将参数kernel = 'linear'替换为kernel = 'rbf'：

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.10, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.show()
```

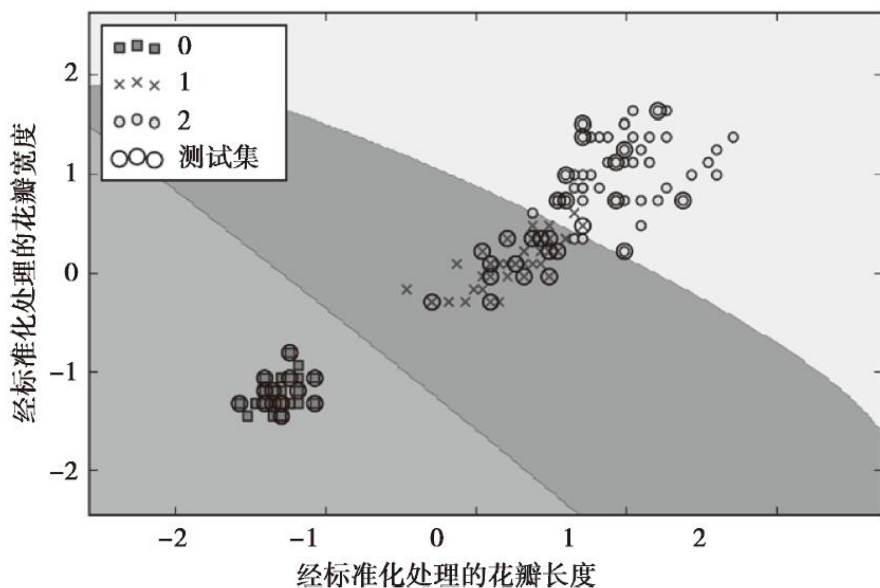
正如结果图像所示，核SVM相对较好地完成了对“异或”数据的划分。



在这里我们将参数γ的值设定为gamma = 0.1，这可以理解为高斯球面的截止参数（cut-off parameter）。如果我们减小γ的值，将会增加受影响的训练样本的范围，这将导致决策边界更加宽松。为了对γ有个更好的直观认识，我们将基于RBF核的SVM应用于鸢尾花数据集。

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

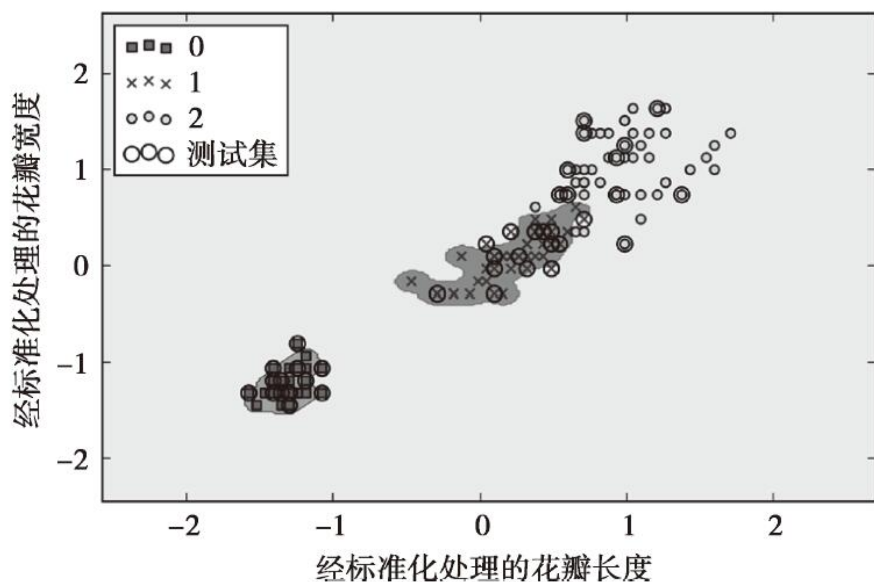
由于我们选择了一个较小的γ值，因此基于RFB核的SVM模型的决策边界就相对宽松，如下图所示：



现在增加γ的值，并观察它对决策边界的影响：

```
>>> svm = SVC(kernel='rbf', random_state=0, gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

通过结果图像可以看到：使用一个相对较大的γ值，使得类别0和1的决策边界紧凑了许多。

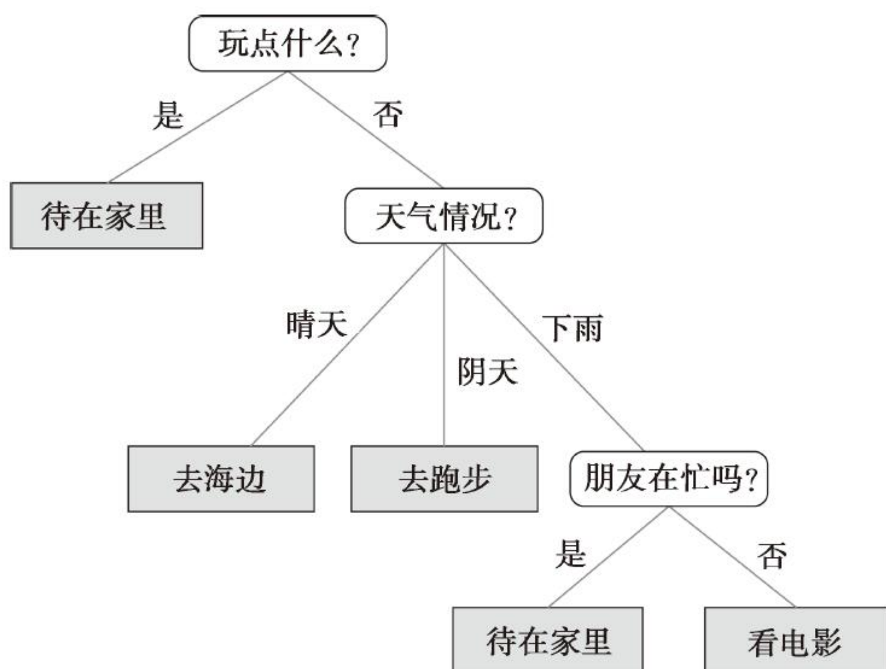


虽然模型对训练数据的拟合非常好，但是类似的分类器对未知数据会有一个较高的泛化误差，这说明对γ的调优在控制过拟合方面也起到了重要作用。

3.6 决策树

决策树吸引人的地方在于其模型的可解释性。正如其名称“决策树”所意味的那样，我们可以把此模型看作将数据自顶向下进行划分，然后通过回答一系列问题来做出决策的方法。

以下图为例，我们使用决策树来决定某一天的活动：



基于训练数据集的特征，决策树模型通过一系列的问题来推断样本的类标。虽然上面的示例在类别变量的基础上给出了决策树的概念，但这些概念在面对特征变量时也同样适用。此模型也适用于数据特征取值为实数

的鸢尾花数据集。例如，我们可以简单为萼片宽度设定一个临界值，并提出一个二元问题："萼片宽度是否达到2.8厘米？"

使用决策树算法，我们从树根开始，基于可获得最大信息增益（information gain，IG）的特征来对数据进行划分，我们将在下一节详细介绍信息增益的概念。通过迭代处理，在每个子节点上重复此划分过程，直到叶子节点。这意味着在每一个节点处，所有的样本都属于同一类别。在实际应用中，这可能会导致生成一棵深度很大且拥有众多节点的树，这样容易产生过拟合问题，由此，我们一般通过对树进行"剪枝"来限定树的最大深度。

3.6.1 最大化信息增益——获知尽可能准确的结果

为了在可获得最大信息增益的特征处进行节点划分，需要定义一个可通过树学习算法进行优化的目标函数。在此，目标函数能够在每次划分时实现对信息增益的最大化，其定义如下：

$$\mathrm{IG}(D_{\mathrm{p}}, f)=I(D_{\mathrm{p}})-\sum_{j=1}^{m}\frac{N_j}{N_{\mathrm{p}}}I(D_j)$$

其中，f为将要进行划分的特征，D_p与D_j分别为父节点和第j个子节点，I为不纯度衡量标准，N_p为父节点中样本的数量，N_j为第j个子节点中样本的数量。可见，信息增益只不过是父节点的不纯度与所有子节点不纯度总和之差——子节点的不纯度越低，信息增益越大。然而，出于简化和缩小组合搜索空间的考虑，大多数（包括scikit-learn）库中实现的都是二叉决策树。这意味着每个父节点被划分为两个子节点：D_{left}和D_{right}。

$$\mathrm{IG}(D_{\mathrm{p}}, a)=I(D_{\mathrm{p}})-\frac{N_{\mathrm{left}}}{N_{\mathrm{p}}}I(D_{\mathrm{left}})-\frac{N_{\mathrm{right}}}{N_{\mathrm{p}}}I(D_{\mathrm{right}})$$

就目前来说，二叉决策树中常用的三个不纯度衡量标准或划分标准分别是：基尼系数（Gini index，I_G）、熵（entropy，I_H），以及误分类率（classification error，I_E）。我们从非空类别（p（i|t）≠0）熵的定义开始讲解：

$$I_{\mathrm{H}}(t)=-\sum_{i=1}^{c}p(i|t)\log_2 p(i|t)$$

其中，p（i|t）为特定节点t中，属于类别c的样本占特定节点t中样本总数的比例。如果某一节点中所有的样本都属于同一类别，则其熵为0，

当样本以相同的比例分属于不同的类时，熵的值最大。例如，对二类别分类来说，当p（i=1|t）=1或p（i=0|t）=0时，熵为0，如果类别均匀分布，即p（i=1|t）=0.5且p（i=1|t）=0.5时，熵为1。因此，我们可以说在决策树中熵的准则就是使得互信息最大化。

直观地说，基尼系数可以理解为降低误分类可能性的标准：

$$I_G(t)=\sum_{i=1}^{c}p(i|t)(-p(i|t))=1-\sum_{i=1}^{c}p(i|t)^2$$

与熵类似，当所有类别是等比例分布时，基尼系数的值最大，例如在二类别分类中（c=2）：

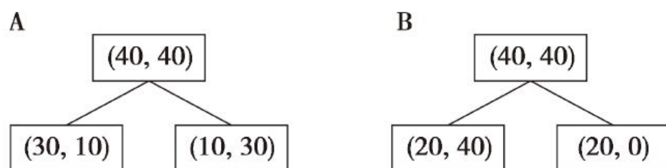
$$1-\sum_{i=1}^{c}0.5^2=0.5$$

然而，在实践中，基尼系数与熵通常会生成非常类似的结果，并不值得花费大量时间使用不纯度标准评估树的好坏，而通常尝试使用不同的剪枝算法。

另一种不纯度衡量标准是误分类率：

$$I_E=1-\max\{p(i|t)\}$$

这是一个对于剪枝方法很有用的准则，但是不建议用于决策树的构建过程，因为它对节点中各类别样本数量的变动不敏感。我们通过下图中两种可能的划分情况对此进行解释：



以数据集D_p为例（在父节点D_p），它包含属于类别1的40个样本和属于类别2的40个样本，我们将它们分别划分到D_{left}和D_{right}。在A和B两种划分情况下，使用误分类率得到的信息增益都是相同的（$IG_E = 0.25$）：

$$I_E(D_p)=1-0.5=0.5$$

$$A:I_E(D_{left})=1-\frac{3}{4}=0.25$$

$$A:I_E(D_{right})=1-\frac{3}{4}=0.25$$

$$A:IG_E=0.5-\frac{4}{8}0.25-\frac{4}{8}0.25=0.25$$

$$B:I_E(D_{left})=1-\frac{4}{6}=\frac{1}{3}$$

$$B:I_E(D_{right})=1-1=0$$

$$B:IG_E=0.5-\frac{6}{8}\times\frac{1}{3}-0=0.25$$

然而，在使用基尼系数时，与划分A（$IG_G = 0.125$）相比，更倾向于使用B（$IG_G = 0.16$）的划分，因为这样子节点处的类别纯度相对更高：

$$I_G(D_p)=1-(0.5^2+0.5^2)=0.5$$

$$A:I_G(D_{left})=1-\left(\left(\frac{3}{4}\right)^2+\left(\frac{1}{4}\right)^2\right)=\frac{3}{8}=0.375$$

$$A:I_G(D_{right})=1-\left(\left(\frac{1}{4}\right)^2+\left(\frac{3}{4}\right)^2\right)=\frac{3}{8}=0.375$$

$$A:I_G=0.5-\frac{4}{8}0.375-\frac{4}{8}0.375=0.125$$

$$B:I_G(D_{left})=1-\left(\left(\frac{2}{6}\right)^2+\left(\frac{4}{6}\right)^2\right)=\frac{4}{9}=0.\dot{4}$$

$$B:I_G(D_{right})=1-(1^2+0^2)=0$$

$$B:IG_G=0.5-\frac{6}{8}0.\bar{4}-0=0.1\bar{\bar{6}}$$

同样，以熵为评估标准时，也是B（IG_G = 0.31）的信息增益大于A（IG_G = 0.19）：

$$I_H(D_p)=-(0.5\log_2(0.5)+0.5\log_2(0.5))=1$$

$$A:I_H(D_{left})=-\left(\frac{3}{4}\log_2\left(\frac{3}{4}\right)+\frac{1}{4}\log_2\left(\frac{1}{4}\right)\right)=0.81$$

$$A:I_H(D_{right})=-\left(\frac{1}{4}\log_2\left(\frac{1}{4}\right)+\frac{3}{4}\log_2\left(\frac{3}{4}\right)\right)=0.81$$

$$A:IG_H=1-\frac{4}{8}\times0.81-\frac{4}{8}\times0.81=0.19$$

$$B:I_H(D_{left})=-\left(\frac{2}{6}\log_2+\left(\frac{2}{6}\right)+\frac{4}{6}\log_2+\left(\frac{4}{6}\right)\right)=0.92$$

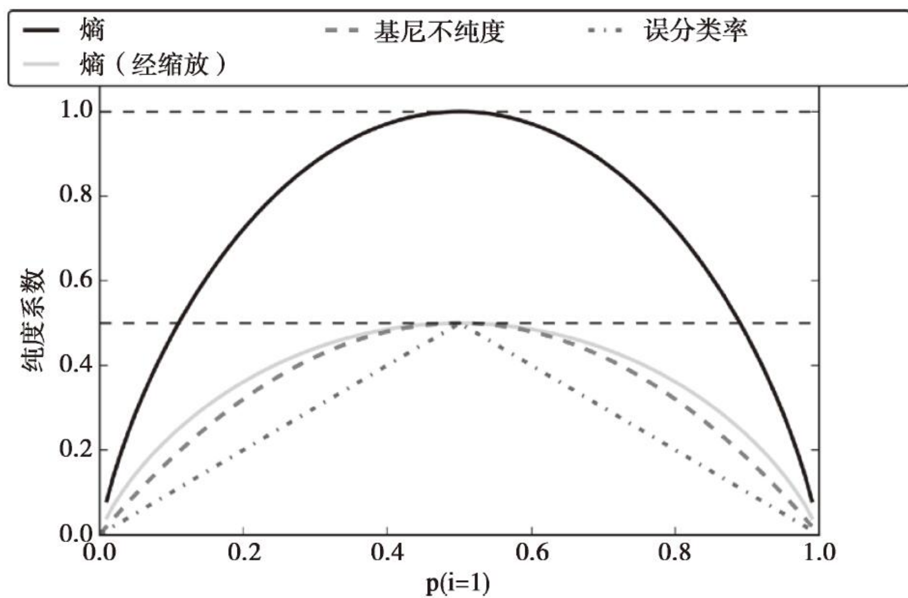
$$B:I_H(D_{right})=0$$

$$B:\mathrm{IG}_H=1-\frac{6}{8}\times0.92-0=0.31$$

为了更直观地比较前面提到的三种不同的不纯度衡量标准，我们绘制样本属于类别1、概率介于[0,1]情况下不纯度的图像。注意，我们在图像中对熵（entropy/2）做了缩放，以便直接观察熵和误分类率对基尼系数的影响。代码如下：

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                   ['Entropy', 'Entropy (scaled)',
...                   'Gini Impurity',
...                   'Misclassification Error'],
...                   ['-', '-', '--', '-.'],
...                   ['black', 'lightgray',
...                      'red', 'green', 'cyan']):
...     line = ax.plot(x, i, label=lab,
...                    linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...           ncol=3, fancybox=True, shadow=False)
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('Impurity Index')
>>> plt.show()
```

执行上述代码后可得到如下图像：



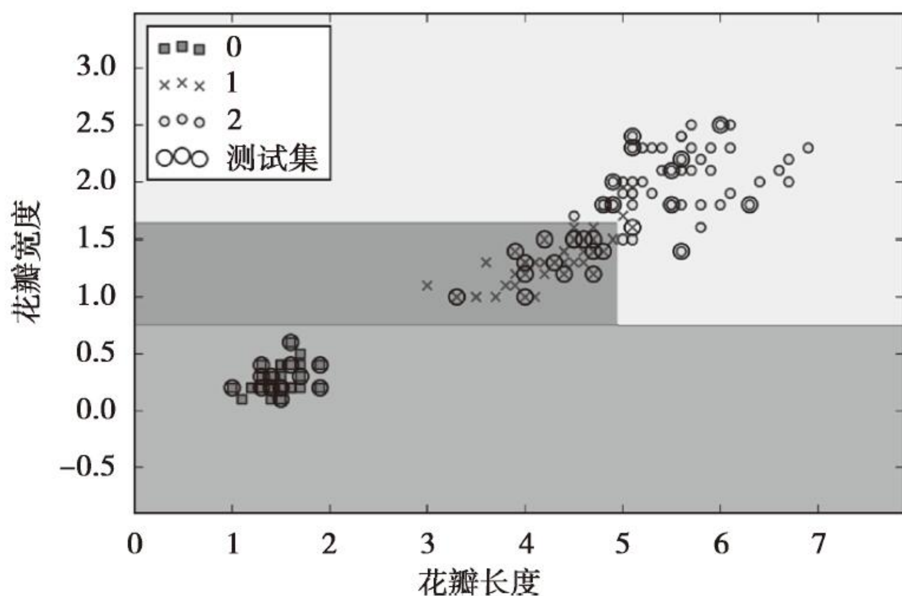
熵
基尼不纯度
误分类率
熵（经缩放）
1.0
0.8
0.6
0.4
0.2
0.0
纯度系数
0.0
0.2
0.4
0.6
0.8
1.0
p(i=1)

3.6.2　构建决策树

决策树可以通过将特征空间进行矩形划分的方式来构建复杂的决策边界。然而，必须注意：深度越大的决策树，决策边界也就越复杂，因而很容易产生过拟合现象。借助于scikit-learn，我们将以熵作为不纯度度量标准，构建一棵最大深度为3的决策树。虽然特征缩放是出于可视化的目的，但在决策树算法中，这不是必须的。代码如下：

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=3, random_state=0)
>>> tree.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined, y_combined,
...                       classifier=tree, test_idx=range(105,150))
>>>plt.xlabel('petal length [cm]')
>>>plt.ylabel('petal width [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

执行上述代码，我们得到了决策数中与坐标轴平行的典型决策边界：



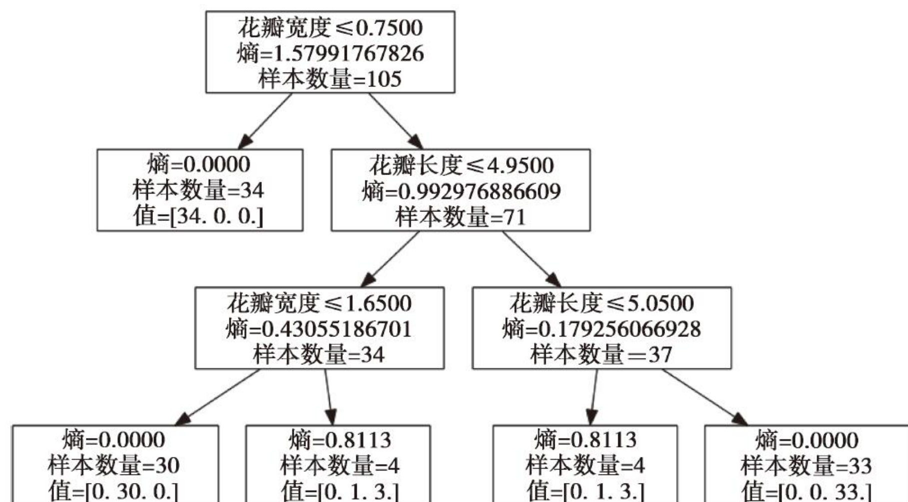
scikit-learn一个吸引人的功能就是，它允许将训练后得到的决策树导出为.dot格式的文件，这使得我们可以使用GraphViz程序进行可视化处理。GraphViz支持Linux、Windows和Mac OS X系统，可在http://www.graphviz.org免费下载。

首先我们从scikit-learn的tree子模块中调用export_graphviz函数生成.dot文件，代码如下：

```
>>> from sklearn.tree import export_graphviz
>>> export_graphviz(tree,
...                     out_file='tree.dot',
...                     feature_names=['petal length', 'petal width'])
```

在电脑上安装好GraphViz后，通过命令行窗口在保存tree.dot文件的路径处执行下面的命令，可将tree.dot文件转换为PNG文件。

```
> dot -Tpng tree.dot -o tree.png
```



通过观察GraphViz创建的图像，可以很好地回溯决策树在训练数据集上对各节点进行划分的过程。最初，决策树根节点包含105个样本，以花瓣宽度≤0.75厘米为界限，将其划分为分别包含34和71个样本的两个子节点。第一次划分后，可以看到：左子树上的样本均来自Iris-Setosa类（熵＝0），已经无需再划分。在右子树上进一步划分，直到将Iris-Versicolor和Iris-Virginica两类分开。

3.6.3　通过随机森林将弱分类器集成为强分类器

由于具备好的分类能力、可扩展性、易用性等特点，随机森林（random forest）过去十年间在机器学习应用领域获得了广泛的关注。直观上，随机森林可以视为多棵决策树的集成。集成学习的基本理念就是将弱分类器集成为鲁棒性更强的模型，即一个能力更强的分类器，集成后具备更好的泛化误差，不易产生过拟合现象。随机森林算法可以概括为四个简单的步骤：

1）使用bootstrap抽样方法随机选择n个样本用于训练（从训练集中随机可重复地选择n个样本）。

2）使用第1）步选定的样本构造一棵决策树。节点划分规则如下：

（1）不重复地随机选择d个特征；

（2）根据目标函数的要求，如最大化信息增益，使用选定的特征对节点进行划分。

3）重复上述过程1～2000次。

4）汇总每棵决策树的类标进行多数投票（majority vote）。多数投票将在第7章中详细介绍。

此处，步骤2）对单棵决策树的训练做了少许修改：在对节点进行最优划分的判定时，并未使用所有的特征，而只是随机选择了一个子集。

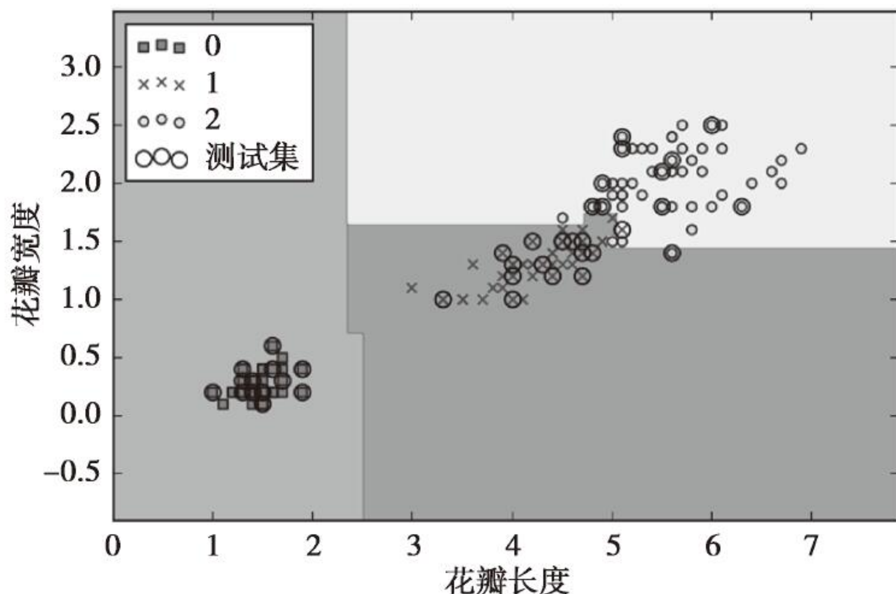
虽然随机森林没有决策树那样良好的可解释性，但其显著的优势在于不必担心超参值的选择。我们通常不需要对随机森林进行剪枝，因为相对于单棵决策树来说，集成模型对噪声的鲁棒性更好。在实践中，我们真正需要关心的参数是为构建随机森林所需的决策树数量（步骤3））。通常情况下，决策树的数量越多，但是随机森林整体的分类表现就越好，但这同时也相应地增加了计算成本。

尽管在实践中不常见，但是随机森林中可优化的其他超参分别是：bootstrap抽样的数量（步骤1））以及在节点划分中使用的特征数量（步骤（2）），它们所采用的技术将在第5章中讨论。通过选择bootstrap抽样中样本数量n，我们可以控制随机森林的偏差与方差权衡的问题。如果n的值较大，就降低了随机性，由此更可能导致随机森林的过拟合。反之，我们可以基于模型的性能，通过选择较小的n值来降低过拟合。包括scikit-learn中RandomForestClassifier在内的大多数对随机森林的实现中，bootstrap抽样的数量一般与原始训练集中样本的数量相同，因为这样在折中偏差与方差方面一般会有一个好的均衡结果。而对于在每次节点划分中用到的特征数量m，我们选择一个比训练集中特征总量小的值。在scikit-learn及其他程序实现中常用的默认值一般是$d=\sqrt{m}$，其中m是训练集中特征的总量。

为了方便起见，我们不是自己从决策树开始构造随机森林，而是使用scikit-learn中已有的实现来完成：

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(criterion='entropy',
...                                 n_estimators=10,
...                                 random_state=1,
...                                 n_jobs=2)
>>> forest.fit(X_train, y_train)
>>> plot_decision_regions(X_combined, y_combined,
...                  classifier=forest, test_idx=range(105,150))
>>> plt.xlabel('petal length')
>>> plt.ylabel('petal width')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

执行上述代码后，通过随机森林中决策树的组合形成的决策区域如下图所示：



在上述代码中，我们在节点划分中以熵为不纯度衡量标准，且通过参数n_estimators使用了10棵决策树进行随机森林的训练。虽然我们只是在一个很小的训练数据集上构建了一个非常小的随机森林，但是出于演示的要求，我们使用n_jobs参数来设定训练过程中所需的处理器内核的数量

（在此使用了CPU的两个内核）。

3.7 惰性学习算法——k-近邻算法

本章将要讨论的最后一个监督学习算法是k-近邻算法（k-nearest neighbor classifier，KNN），此算法非常有趣，因为它区别于目前已经介绍过的所有学习算法。

KNN是惰性学习算法的典型例子。说它具有惰性不是因为它看起来简单，而是因为它仅仅对训练数据集有记忆功能，而不会从训练集中通过学习得到一个判别函数。



参数化模型与非参数化模型

机器学习算法可以划分为参数化模型和非参数化模型。当使用参数化模型时，需要我们通过训练数据估计参数，并通过学习得到一个模式，以便在无需原始训练数据信息的情况下对新的数据点进行分类操作。典型的参数化模型包括：感知器、逻辑斯谛回归和线性支持向量机等。与之相反，非参数化模型无法通过一组固定的参数来进行表征，而参数的数量也会随着训练数据的增加而递增。我们已经学习了两个非参数化模型：决策树（包括随即森林）和核SVM。

KNN属于非参数化模型的一个子类，它可以被描述为基于实例的学习（instance-based learning）。此类模型的特点是会对训练数据进行记忆；而惰性学习（lazy learning）则是基于实例学习的一个特例，它在学习阶段的计算成本为0。

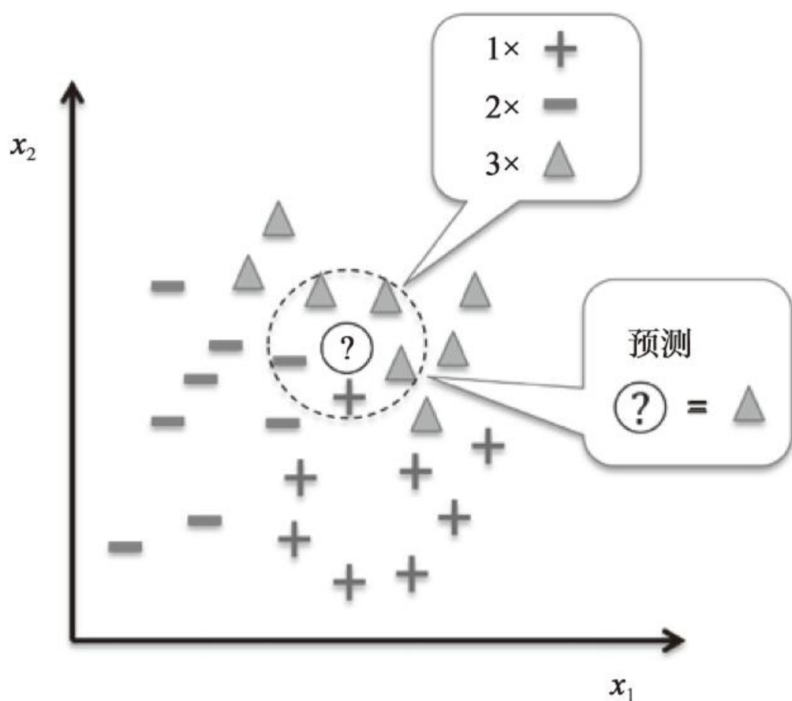
KNN算法本身是很简单的，可以归纳为以下几步：

1）选择近邻的数量k和距离度量方法。

2）找到待分类样本的k个最近邻居。

3）根据最近邻的类标进行多数投票。

下图说明了新的数据点（问号所在位置）如何根据最近的5个邻居进行多数投票而被标记上三角形类标：



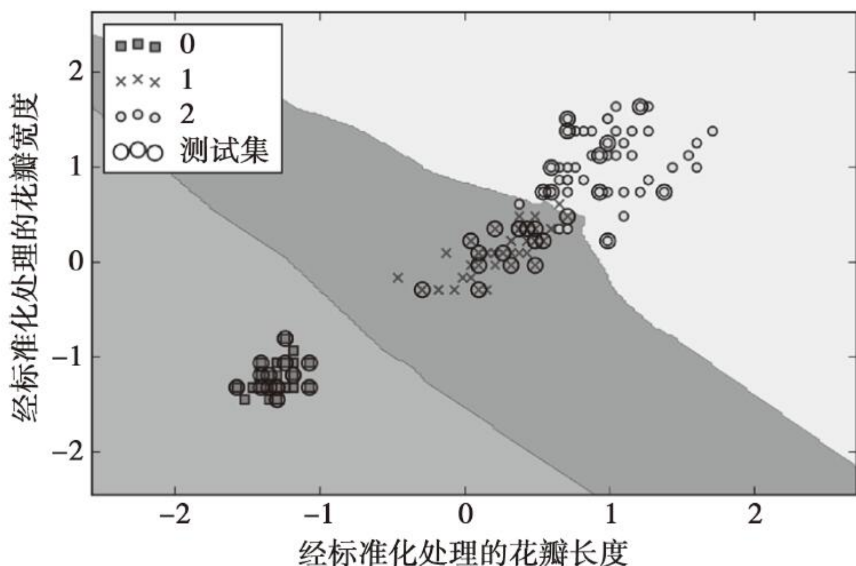
基于选定的距离度量标准，KNN算法从训练数据集中[1]找到与待预测目标点的k个距离最近的样本（最相似）。目标点的类标基于这k个最近的邻居的类标使用多数投票确定。

这种基于记忆的学习算法的优点在于：分类器可以快速地适应新的训练数据。不过其缺点也是显而易见的：在最坏情况下，计算复杂度随着样本数量的增多而呈线性增长，除非数据集中的样本维度（特征数量）有限，而且使用了高效的数据结构（如KD树等）。具体请参照J.H.Friedman、J.L.Bentley在R.A.Finkel发表在ACM Transactions on Mathematical Software（TOMS）3（3）:209–226,1977的论文“An algorithm for finding best matches in logari-thmic expected time”。此外，我们还不能忽视训练样本，因为此模型没有训练的步骤。由此一来，如果使用了大型数据集，对于存储空间来说也是一个挑战。

使用以下代码，以欧几里得距离为度量标准，使用scikit-learn实现了一个KNN模型。

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                            metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                       classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.show()
```

将此数据集上的KNN模型的近邻数量设定为5个，我们得到了相对平滑的决策边界，如下图所示：



通常情况下，scikit-learn中实现的KNN算法对类标的判定倾向于与样本距离最近邻居的类标。如果几个邻居的距离相同，则判定为在训练数据集中位置靠前的那个样本的类标。

就KNN来说，找到正确的k值是在过拟合与欠拟合之间找到平衡的关键所在。我们还必须保证所选的距离度量标准适用于数据集中的特征。相对简单的欧几里得距离度量标准常用于特征值为实数的样本，如鸢尾花数据集中的花朵，其特征值是以厘米为单位的实数。注意，当我们使用欧几里得距离时，对数据进行标准化处理，保持各属性度量的尺度统一也是非常重要的。在代码中用到的“闵可夫斯基”（'minkowski'）距离是对欧几里得距离及曼哈顿距离的一种泛化，可写作：

$$d(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(i)})=\sqrt[p]{\sum_k |x_k^{(i)} x_k^{(j)}|^p}$$

如果将参数设定为p = 2，则为欧几里得距离；当p = 1时，就是曼哈顿

距离。scikit-learn中还实现了许多其他的距离度量标准，具体内容可见如下网址：http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html。

维度灾难

由于维度灾难（curse of dimensionality）的原因，使得KNN算法易于过拟合。维度灾难是指这样一种现象：对于一个样本数量大小稳定的训练数据集，随着其特征数量的增加，样本中有具体值的特征数量变得极其稀疏（大多数特征的取值为空）。直观地说，可以认为即使是最近的邻居，它们在高维空间中的实际距离也是非常远的，因此难以给出一个合适的类标判定。

在逻辑斯谛回归一节中，我们讨论了通过正则化使其避免过拟合。不过，正则化方法并不适用于诸如决策树和KNN等算法，但可以使用特征选择和降维等技术来帮助其避免维度灾难。具体内容将在下一章中探讨。

[1] 此处不应叫训练数据集，与协同过滤类似，就是数据集，但原文如此。——译者注

本章小结

在本章中，我们学到了许多不同的机器学习算法，可用于处理线性或者非线性问题。如果关注算法的可解释性，那么决策树是一种特别具有吸引力的算法。逻辑斯谛回归不仅是可以通过梯度下降进行优化的一种有用的在线学习方法，而且可以给出待预测问题可能发生的概率。虽然支持向量机是一种强大的线性模型，而且可以通过核技巧扩展到非线性问题，但是为了达到好的预测效果，它需要调整众多的参数。而集成方法（如随机森林）不需要调整众多的参数且不像决策树那样容易产生过拟合现象，因此在解决实际问题中成为常用的一个模型。作为惰性学习算法，k-近邻算法使得我们在分类领域可以尝试另外一种方式，它不是通过训练模型来进行预测，而更多的是通过计算来完成。

但是，比选择合适的学习算法更重要的是：训练数据集中有什么样的可用数据。任何算法都无法使用缺乏翔实、无歧义的特征而获得好的预测结果。

在下一章，我们将讨论数据预处理、特征选择，以及降维等相关的重要内容，这些都是构建优秀的机器学习模型所必需的。在后续的第6章中，我们将学到如何评估及比较模型的性能，并学习一些有用的、微调不同算法的技巧。

第4章　数据预处理——构建好的训练数据集

机器学习算法最终学习结果的优劣取决于两个主要因素：数据的质量和数据中蕴含的有用信息的数量。因此，在将数据集应用于学习算法之前，对其进行检验及预处理是至关重要的。在本章中，我们将讨论主要的数据预处理技术，使用这些技术可以高效地构建好的机器学习模型。

本章将涵盖如下主题：

·数据集中缺失数据的删除和填充

·数据格式化

·模型构建中的特征选择

4.1 缺失数据的处理

在实际应用过程中，样本由于各种原因缺少一个或多个值的情况并不少见。其原因主要有：数据采集过程中出现了错误，常用的度量方法不适用于某些特征，或者在调查过程中某些数据未被填写，等等。通常，我们见到的缺失值是数据表中的空值，或者是类似于NaN（Not A Number，非数字）的占位符。

遗憾的是，如果我们忽略这些缺失值，将导致大部分的计算工具无法对原始数据进行处理，或者得到某些不可预知的结果。因此，在做更深入的分析之前，必须对这些缺失值进行处理。而在讨论处理缺失值的几个技巧之前，我们先通过一个CSV（comma-separed values，以逗号为分隔符的数值）文件构造一个简单的例子，以便更好地把握问题的核心所在：

```
>>> import pandas as pd
>>> from io import StringIO
>>> csv_data = '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 0.0,11.0,12.0,'''
>>> # If you are using Python 2.7, you need
>>> # to convert the string to unicode:
>>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>>> df
   A   B    C    D
0  1   2    3    4
1  5   6  NaN    8

2  0  11   12  NaN
```

在上述代码中，我们通过read_csv函数将CSV格式的数据读取到pandas的数据框（DataFrame）中，可以看到：其中有两个缺失值由NaN

替代。示例代码中的StringIO函数在此仅起到演示作用：如果我们的数据是存储在硬盘上的CSV文件，就可以通过此函数以字符串的方式从文件中读取数据，并将其转换成DataFrame的格式赋值给csv_data。

对于大的DataFrame来说，手工搜索缺失值是极其繁琐的，在此情况下，我们可以使用代码中的isnull方法返回一个布尔型的DataFrame值，若DataFrame的元素单元包含数字型数值则返回值为假（False），若数据值缺失则返回值为真（True）。通过sum方法，我们可以得到如下所示的每列中缺失值数量：

```
>>> df.isnull().sum()
A     0
B     0
C     1
D     1
dtype: int64
```

通过这种方式，我们可以得到每列中缺失值的数量。在下面的小节中，我们将学习几种不同的处理缺失数据的策略。

虽然scikit-learn是在NumPy数组的基础上开发的，但有时使用pandas的DataFrame来处理数据会更为方便。在我们使用scikit-learn处理数据之前，可以通过DataFrame的value属性来访问相关的NumPy数组：

```
>>> df.values
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  nan,   8.],
       [ 10.,  11.,  12.,  nan]])
```

4.1.1　将存在缺失值的特征或样本删除

处理缺失数据最简单的方法就是：将包含确实数据的特征（列）或者样本（行）从数据集中删除。可通过dropna方法来删除数据集中包含缺失值的行：

```
>>> df.dropna()
   A  B  C  D
0  1  2  3  4
```

类似地，我们可以将axis参数设为1，以删除数据集中至少包含一个NaN值的列：

```
>>> df.dropna(axis=1)
   A   B
0  1   2
1  5   6
2  0  11
```

dropna方法还支持其他参数，以应对各种缺失值的情况：

```
# only drop rows where all columns are NaN
>>> df.dropna(how='all')

# drop rows that have not at least 4 non-NaN values
>>> df.dropna(thresh=4)

# only drop rows where NaN appear in specific columns (here: 'C')
>>> df.dropna(subset=['C'])
```

删除缺失数据看起来像是一种非常方便的方法，但也有一定的缺点，如：我们可能会删除过多的样本，导致分析结果可靠性不高。从另一方面讲，如果删除了过多的特征列，有可能会面临丢失有价值信息的风险，而这些信息是分类器用来区分类别所必需的。在下一节，我们将学习另外一种最常用的处理缺失数据的方法：插值技术。

4.1.2　缺失数据填充

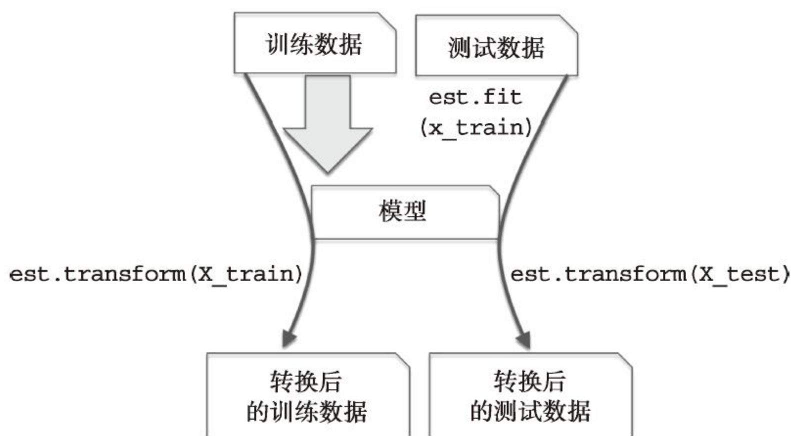
通常情况下，删除样本或者删除数据集中的整个特征列是不可行的，因为这样可能会丢失过多有价值的数据。在此情况下，我们可以使用不同的插值技术，通过数据集中其他训练样本的数据来估计缺失值。最常用的插值技术之一就是均值插补（meaneinputation），即使用相应的特征均值来替换缺失值。我们可使用scikit-learn中的Impute类方便地实现此方法，代码如下：

```
>>> from sklearn.preprocessing import Imputer
>>> imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imr = imr.fit(df)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
array([[  1.,    2.,    3.,    4.],
       [  5.,    6.,    3.,    8.],
       [ 10.,   11.,   12.,    4.]])
```

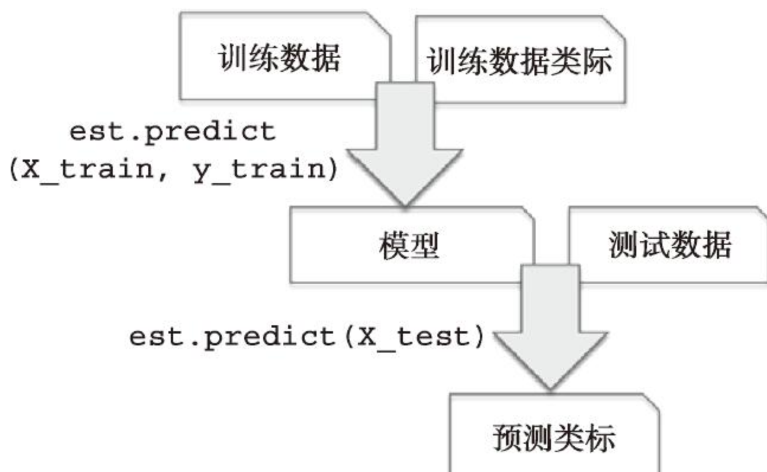
在此，首先计算各特征列的均值，然后使用相应的特征均值对NaN进行替换。如果我们把参数axis = 0改为axis = 1，则用每行的均值来进行相应的替换。参数strategy的可选项还有median或者most_frequent，后者代表使用对应行或列中出现频次最高的值来替换缺失值，常用于填充类别特征值。

4.1.3　理解scikit-learn预估器的API

在上一节中，我们使用了scikit-learn中的Imputer类来填充我们数据集中的缺失值。Imputer类属于sciait-learn中的转换器类，主要用于数据的转换。这些类中常用的两个方法是fit和transform。其中，fit方法用于对数据集中的参数进行识别并构建相应的数据补齐模型，而transform方法则使用刚构建的数据补齐模型对数据集中相应参数的缺失值进行补齐。所有待补齐数据的维度应该与数据补齐模型中其他数据的维度相同。下图解释了转换器类如何对训练和测试数据进行数据处理（在此是数据补齐）：



我们在第3章中用到了分类器，它们在scikit-learn中属于预估器类别，其API的设计与转换器类非常相似。在后续内容中我们将看到，预估器类包含一个predict方法，同时也包含一个transform方法。读者应该记得，我们在训练预估器用于分类任务时，同样使用了一个fit方法来对参数进行设定。在监督学习中，我们额外提供了类标用于构建模型，而模型可通过predict方法对新的样本数据展开预测，如下图所示：


训练数据
训练数据类际
est.predict
(X_train, y_train)
模型
测试数据
est.predict(X_test)
预测类标

4.2 处理类别数据

到目前为止，我们仅学习了处理数值型数据的方法。然而，在真实数据集中，经常会出现一个或多个类别数据的特征列。我们在讨论类别数据时，又可以进一步将他们划分为标称特征（nominal feature）和有序特征（ordinal feature）。可以将有序特征理解为类别的值是有序的或者是可以排序的。如T恤衫的尺寸就是一个有序特征，因为我们可以为其值排序XL＞L＞M。相反，标称数据则不具备排序的特性。继续刚才。例子，我们可以将T恤衫的颜色看作一个标称特征，因为一般说来，对颜色进行比较，如红色大于蓝色这种说法是不符合常理的。

在探索类别数据的处理技巧之前，我们先构造一个数据集来用来描述问题：

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...             ['green', 'M', 10.1, 'class1'],
...             ['red', 'L', 13.5, 'class2'],
...             ['blue', 'XL', 15.3, 'class1']])
>>> df.columns = ['color', 'size', 'price', 'classlabel']
>>> df
   color size  price classlabel
0  green    M   10.1     class1
1    red    L   13.5     class2
2   blue   XL   15.3     class1
```

从代码的输出结果中可以看到，我们新构造的DataFrame分别包含一个标称特征（颜色）、一个有序特征（尺寸）以及一个数值特征（价格）。类标（此处假定我们构造的数据集用于监督学习）存储在最后一列。在本书中，我们讨论的分类学习算法中均不使用有序信息作为类标。

4.2.1　有序特征的映射

为了确保学习算法可以正确地使用有序特征，我们需要将类别字符串转换为整数。但是，没有一个适当的方法可以自动将尺寸特征转换为正确的顺序。由此，需要我们手工定义相应的映射。在接下来的例子中，假设我们了解特征值间的差异，如：XL = L + 1 = M + 2。

```
>>> size_mapping = {
...                  'XL': 3,
...                  'L': 2,
...                  'M': 1}
>>> df['size'] = df['size'].map(size_mapping)
>>> df
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```

如果在后续过程中需要将整数值还原为有序字符串，可以简单地定义一个逆映射字典：inv_size_mapping = {v:k for k,v in size_mappping.items()}，与前面用到的size_mapping类似，可以通过pandas的map方法将inv_size_mapping应用于经过转换的特征列上。

4.2.2 类标的编码

许多机器学习库要求类标以整数值的方式进行编码。虽然scikit-learn中大多数分类预估器都会在内部将类标转换为整数，但通过将类标转换为整数序列能够从技术角度避免某些问题的产生，在实践中这被认为是一个很好的做法。为了对类标进行编码，可以采用与前面讨论的有序特征映射相类似的方式。要清楚一点，类标并不是有序的，而且对于特定的字符串类标，赋予哪个整数值给它对我们来说并不重要。因此，我们可以简单地以枚举的方式从0开始设定类标。

```
>>> import numpy as np
>>> class_mapping = {label:idx for idx,label in
...                  enumerate(np.unique(df['classlabel']))}
>>> class_mapping
{'class1': 0, 'class2': 1}
```

接下来，我们可以使用映射字典将类标转换为整数：

```
>>> df['classlabel'] = df['classlabel'].map(class_mapping)
>>> df
   color  size  price  classlabel
0  green     1   10.1           0
1    red     2   13.5           1
2   blue     3   15.3           0
```

我们可以通过下列代码将映射字典中的键一值对倒置，以将换转过的类标还原回原始的字符串表示：

```
>>> inv_class_mapping = {v: k for k, v in class_mapping.items()}
>>> df['classlabel'] = df['classlabel'].map(inv_class_mapping)
>>> df
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```

此外，使用scikit-learn中的LabelEncoder类可以更加方便地完成对类标的整数编码工作：

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([0, 1, 0])
```

请注意：fit_transform方法相当于分别调用fit和transform方法的快捷方式，我们还可以使用inverse_transform方法将整数类标还原为原始的字符串表示：

```
>>> class_le.inverse_transform(y)
array(['class1', 'class2', 'class1'], dtype=object)
```

4.2.3 标称特征上的独热编码

在前面小节中，我们曾使用字典映射的方法将有序的尺寸特征转换为整数。由于scikit-learn的预估器将类标作为无序数据进行处理，可以使用scikit-learn中的LabelEncoder类将字符串类标转换为整数。同样，也可以用此方法处理数据集中标称数据格式的color列，代码如下：

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

执行上述代码，NumPy数组X的第一列现在被赋予新的color值，具体编码结果如下：

·blue→0

·green→1

·red→2

如果我们就此打住，直接将得到的数组导入分类器，那么就会犯处理类别数据时最常见的错误。读者发现问题所在了吗？虽然颜色的值并没有特定的顺序，但是学习算法将假定green大于blue、red大于green。虽然算法的这一假定不太合理，但最终还是能够生成有用的结果。然而，这个结果可能不是最优的。

解决此问题最常用的方法就是独热编码（one-hot encoding）技术。这种方法的理念就是创建一个新的虚拟特征（dummy feature），虚拟特征的每一列各代表标称数据的一个值。在此，我们将color特征转换为三个新的特征：blue、green以及red。此时可以使用二进制值来标识样本的颜色。例如，蓝色样本可以标识为：blue = 1，green = 0，以及red = 0。此编码转换操作可以使用scikit-learn.proprocessing模块中的OneHotEncoder来实现：

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> ohe = OneHotEncoder(categorical_features=[0])
>>> ohe.fit_transform(X).toarray()
array([[  0. ,    1. ,    0. ,    1. ,   10.1],
       [  0. ,    0. ,    1. ,    2. ,   13.5],
       [  1. ,    0. ,    0. ,    3. ,   15.3]])
```

当我们初始化OneHotEncoder对象时，需要通过categorical_features参数来选定我们要转换的特征所在的位置（如color是特征矩阵X的第1列）。默认情况下，当我们调用OneHotEncoder的transform方法时，它会返回一个稀疏矩阵。出于可视化的考虑，我们可以通过toarray方法将其转换为一个常规的NumPy数组。稀疏矩阵是存储大型数据集的一个有效方法，被许多scikit-learn函数所支持，特别是在数据包含很多零值时非常有用。为了略过toarry的使用步骤，我们也可以通过在初始化阶段使用OneHotEncoder（...,sparse = False）来返回一个常规的Numpy数组。

另外，我们可以通过pandas中的get_dummies方法，更加方便地实现独热编码技术中的虚拟特征。当应用于DataFrame数据时，get_dummies方法只对字符串列进行转换，而其他的列保持不变。

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
   price  size  color_blue  color_green  color_red
0   10.1     1           0            1          0
1   13.5     2           0            0          1
2   15.3     3           1            0          0
```

4.3 将数据集划分为训练数据集和测试数据集

在第1章和第3章中，我们简单介绍了将数据集划分为训练数据集和测试数据集的相关内容。请记住，我们可以将测试数据集理解为模型投诸于现实应用前的最终测试集。在本小节，我们准备了一个新的数据集，葡萄酒数据集（Wine dataset）。在完成数据集的预处理后，我们将探讨不同的特征选择技巧以降低数据集的维度。

葡萄酒数据集是另一个开源数据集，可以通过UCI机器学习样本数据库获得（https://archive.ics.uci.edu/ml/datasets/Wine），它包含178个葡萄酒样本，每个样本通过13个特征对其化学特征进行描述。

借助于pandas库，可以直接从UCI机器学习样本数据库中在线读取开源的葡萄酒数据集。

```
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash', 'Magnesium',
...                    'Total phenols', 'Flavanoids',
...                    'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>>  print('Class labels', np.unique(df_wine['Class label']))
Class labels [1 2 3]
>>> df_wine.head()
```

葡萄酒样本库通过13个不同的特征，对178个葡萄酒样本的化学特征做出描述，如下表：

	Class label	Alcohol	Malic acid	Ash	Alcalinity of adh	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

这些样本属于类标分别为1、2、3的三个不同的类别，对应于三种生长在意大利不同地区的葡萄种类。

想要将此数据集随机划分为测试数据集和训练数据集，一个简便方法是使用scikit-learn下cross_validation子模块中的train_test_split函数：

```
>>> from sklearn.cross_validation import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...         train_test_split(X, y, test_size=0.3, random_state=0)
```

首先，将NumPy数组中的第1～13个特征列赋值给变量X，将第1列中的类标赋值给变量y。然后，使用train_test_split函数随机地将X和y各自划分为训练数据集和测试数据集。设定test_size = 0.3，可以将30%的样本划分到X_test和y_test，剩余的70%样本划分到X_train及y_train。

如果要将数据集划分为训练和测试数据集，必须牢记：尽量保留有价值的信息，这些信息将有利于训练机器学习算法。因此，我们一般不会为测试数据集分配太多的数据。不过，测试集越小，对泛化误差的估计将会越不准确。在对数据集进行划分时，需要对此进行权衡。在实际应用中，基于原始数据的大小，常用的划分比例是60:40、70:30，或者80:20。对于非常庞大的数据集，将训练集和测试集的比例按照90:10或者99:1进行划分也是常见且可以接受的。为了让模型获得最佳的性能，完成分类模型的测试后，通常在整个数据集上需再次对模型进行训练。

4.4　将特征的值缩放到相同的区间

特征缩放（peature scaling）是数据预处理过程中至关重要的一步，但却极易被人们忽略。决策树和随机森林是机器学习算法中为数不多的不需要进行特征缩放的算法。然而，对大多数机器学习和优化算法而言，将特征的值缩放到相同的区间可以使其性能更佳，例如在第2章实现的梯度下降（gradient descent）优化算法。

特征缩放的重要性可以通过一个简单的例子来描述。假定我们有两个特征：一个特征值的范围为1～10；另一个特征值的范围为1～100000。回忆一下第2章中Adaline的平方误差函数，直观地说，算法将主要根据第二个特征上较大的误差进行权重的优化。另外还有k-近邻（k-nearest neighbor，KNN）算法，它以欧几里得距离作为相似性度量，样本间距离的计算也以第二个特征为主。

目前，将不同的特征缩放到相同的区间有两个常用的方法：归一化（normalization）和标准化（standardization）。这两个词在不同的领域中使用较为宽松，其含义由具体语境所确定。多数情况下，归一化指的是将特征的值缩放到区间[0,1]，它是最小-最大缩放的一个特例。为了对数据进行规范化处理，我们可以简单地在每个特征列上使用min-max缩放，通过如下公式可以计算出一个新的样本$x^{(i)}$的值$x_{norm}^{(i)}$：

$$x_{norm}^{(i)}=\frac{x^{(i)}-x_{\min}}{x_{\max}-x_{\min}}$$

其中，$x^{(i)}$是一个特定的样本，x_{min}和x_{max}分别是某特征列的最小

值和最大值。

在scikit-learn中，已经实现了最小-最大缩放，使用方法如下：

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

当遇到需将数值限定在一个有界区间的情况时，我们常采用最小-最大缩放来进行有效的规范化。但在大部分机器学习算法中，标准化的方法却更加实用。这是因为：许多线性模型，如第3章中讨论过的逻辑斯谛回归和支持向量机，在对它们进行训练的最初阶段，即权重初始化阶段，可将其值设定0或是趋近于0的随机极小值。通过标准化，我们可以将特征列的均值设为0，方差为1，使得特征列的值呈标准正态分布，这更易于权重的更新。此外，与最小-最大缩放将值限定在一个有限的区间不同，标准化方法保持了异常值所蕴含的有用信息，并且使得算法受到这些值的影响较小。

标准化的过程可用如下方程表示：

$$x_{std}^{(i)}=\frac{x^{(i)}-\mu_x}{\sigma_x}$$

其中，μ_x和σ_x分别为样本某个特征列的均值和标准差。

在包含值为0~5的数据样本上，采用标准化和归一化两种常用的技术进行特征缩放，其两者之间的区别如下表所示：

输　入	标　准　化	归　一　化
0.0	−1.336306	0.0
1.0	−0.801784	0.2
2.0	−0.267261	0.4
3.0	0.267261	0.6
4.0	0.801784	0.8
5.0	1.336306	1.0

与MinMaxScaler类似，scikit-learn也已经实现了标准化类：

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

需要再次强调的是：我们只是使用了StandardScaler对训练数据进行拟合，并使用相同的拟合参数来完成对测试集以及未知数据的转换。

4.5 选择有意义的特征

如果一个模型在训练数据集上的表现比在测试数据集上好很多，这意味着模型过拟合（overfitting）于训练数据。过拟合是指模型参数对于训练数据集的特定观测值拟合得非常接近，但训练数据集的分布与真实数据并不一致——我们称之为模型具有较高的方差。产生过拟合的原因是建立在给定训练数据集上的模型过于复杂，而常用的降低泛化误差的方案有：

1）收集更多的训练数据

2）通过正则化引入罚项

3）选择一个参数相对较少的简单模型

4）降低数据的维度

一般来说，收集更多的训练数据不太适用。在下一章中，我们将探讨一种更为有用的技术，以验证收集更多训练数据是否能够对解决过拟合问题有所帮助。而在本章的后续内容中，我们将讨论正则化和特征选择降维这两种常用的减少过拟合问题的方法。

4.5.1 使用L1正则化满足数据稀疏化

回顾一下第3章的内容，L2正则化是通过对大的权重增加罚项以降低模型复杂度的一种方法，其中，权重向量w的L2范数如下：

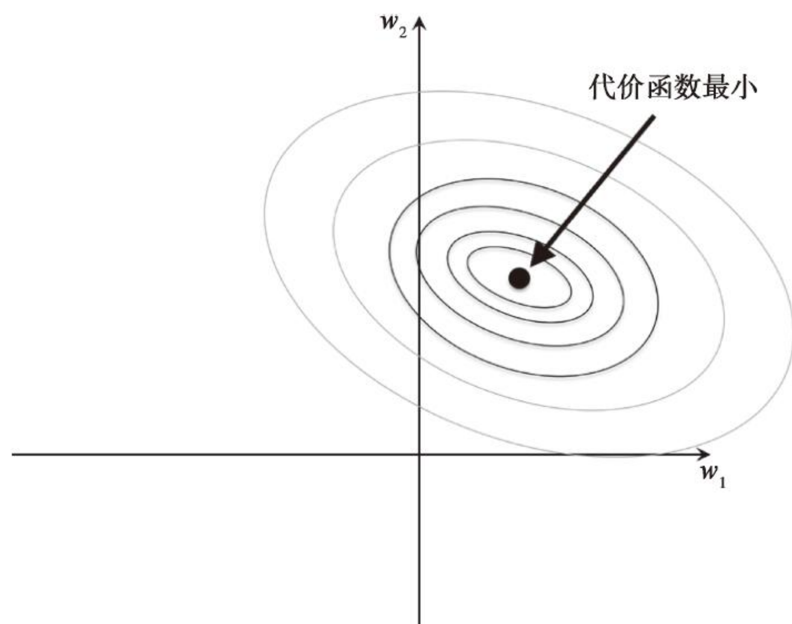
$$L2: \|\boldsymbol{w}\|_2^2 = \sum_{j=1}^{m} w_j^2$$

另外一种降低模型复杂度的方法则与L1正则化相关：

$$L1: \|\boldsymbol{w}\|_1 = \sum_{j=1}^{m} |w_j|$$

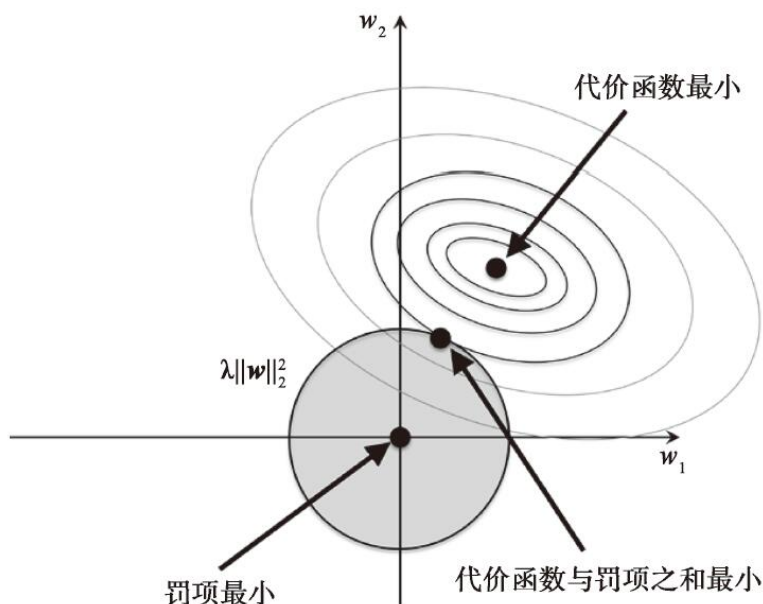
在此，我们只是简单地将权重的平方和用权重绝对值的和来替代。与L2正则化不同，L1正则化可生成稀疏的特征向量，且大多数的权值为0。当高维数据集中包含许多不相关的特征，尤其是不相关的特征数量大于样本数量时，权重的稀疏化处理能够发挥很大的作用。从这个角度来看，L1正则化可以被视作一种特征选择技术。

为了更好地理解L1正则化如何对数据进行稀疏化，我们先退一步，来看一下正则化的几何解释。我们先绘制权重系数w_1和w_2的凸代价函数的等高线。在此，我们将使用第2章中Adaline算法所采用的代价函数-误差平方和（sum of the squared errors，SSE）作为代价函数，因为它的图像是对称的，且比逻辑斯谛回归的代价函数更易于绘制。在后续内容中，我们还将再次使用此代价函数。请记住，我们的目标是找到一个权重系数的组合，能够使得训练数据的代价函数最小，如下图所示（椭圆中心的圆点）：



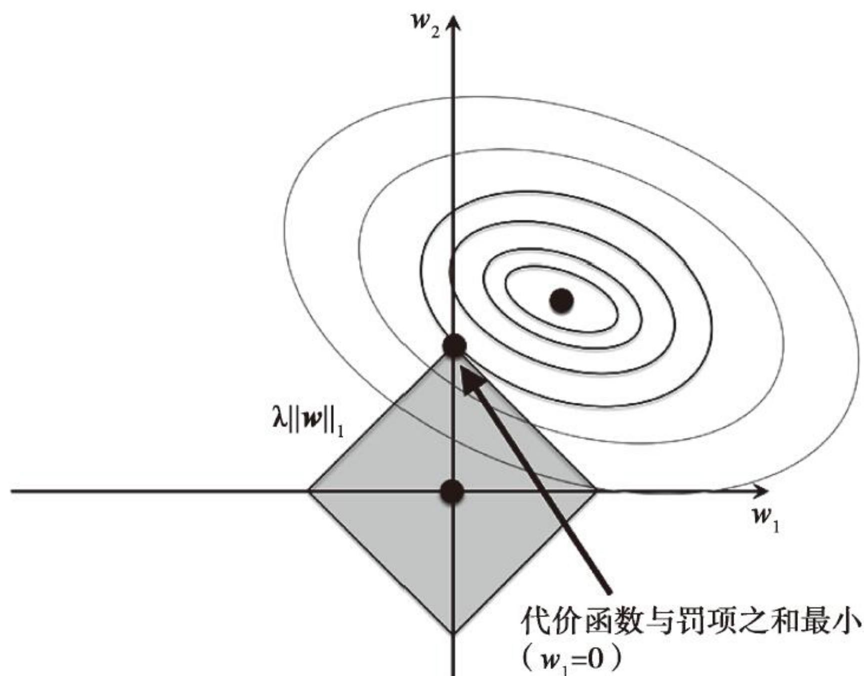
至此，我们可以将正则化看作是在代价函数中增加一个罚项，以对小的权重做出激励，换句话说，我们对大的权重做出了惩罚。

这样，通过正则化参数λ来增加正则化项的强度，使得权重向0收缩，就降低了模型对训练数据的依赖程度。我们使用下图来阐述L2罚项的概念。



我们使用阴影表示的球代表二次的L2正则化项。在此，我们的权重系数不能超出正则化的区域——权重的组合不能落在阴影以外的区域。另一方面，我们仍然希望能够使得代价函数最小化。在罚项的约束下，最好的选择就是L2球与不含有罚项的代价函数等高线的切点。正则化参数λ的值越大，含有罚项的代价函数增长得就越快，L2的球就会越小。例如，如果增大正则化参数的值，使之趋向于无穷大，权重系数将趋近与0，即图像上所示的L2球的圆心。总结一下这个例子的主要内容：我们的目标是使得代价函数和罚项之和最小，这可以理解为通过增加偏差使得模型趋向于简单，以降低在训练数据不足的情况下拟合得到的模型的方差。

现在来讨论L1正则化与稀疏问题。L1正则化的主要理念与我们之前所讨论的相类似。不过，由于L1的罚项是权重系数绝对值的和（请记住L2罚项是二次的），我们可以将其表示为菱形区域，如下图所示：



在上图中，代价函数等高线与L1菱形在$w_1 = 0$处相交。由于L1的边界不是圆滑的，这个交点更有可能是最优的——也就是说，椭圆形的代价函数边界与L1菱形边界的交点位于坐标轴上，这也就使得模型更加稀疏。L1正则化如何导致数据稀疏的数学论证已超出了本书的范围。如果你感兴趣，请参阅Trevor Hastie、Robert Tibshirani和Jerome Friedman的著作The Elements of Statistical Learning中的3.4节，里面有关于L2与L1正则化的精彩论述。

对于scikit-learn中支持L1的正则化模型，我们可以通过将penalty参数设定为'l1'来进行简单的数据稀疏处理：

```
>>> from sklearn.linear_model import LogisticRegression
>>> LogisticRegression(penalty='l1')
```

将其应用于经过标准化处理的葡萄酒数据，经过L1正则化的逻辑斯谛回归模型可以产生如下稀疏化结果：

```
>>> lr = LogisticRegression(penalty='l1', C=0.1)
>>> lr.fit(X_train_std, y_train)
>>> print('Training accuracy:', lr.score(X_train_std, y_train))
Training accuracy: 0.983870967742
>>> print('Test accuracy:', lr.score(X_test_std, y_test))
Test accuracy: 0.981481481481
```

训练和测试的精确度（均为98%）显示此模型未出现过拟合。通过lr.intercept_属性得到截距项后，可以看到代码返回了包含三个数值的数组：

```
>>> lr.intercept_
array([-0.38379237, -0.1580855 , -0.70047966])
```

由于我们在多类别分类数据集上使用了LogisticRegression对象，它默认使用一对多（One-vs-Rest，OvR）的方法。其中，第一个截距项为类别1相对于类别2、3的匹配结果，第二个截距项为类别2相对于类别1、3的匹配结果，第三个截距项则为类别3相对于类别1、2的匹配结果。

```
>>> lr.coef_
array([[ 0.280, 0.000, 0.000, -0.0282, 0.000,
         0.000, 0.710, 0.000, 0.000, 0.000,
         0.000, 0.000, 1.236],
       [-0.644, -0.0688 , -0.0572, 0.000, 0.000,
         0.000, 0.000, 0.000, 0.000, -0.927,
         0.060, 0.000, -0.371],
       [ 0.000, 0.061, 0.000, 0.000, 0.000,
         0.000, -0.637, 0.000, 0.000, 0.499,
        -0.358, -0.570, 0.000
       ]])
```

通过lr.coef_属性得到的权重数组包含三个权重系数向量，每一个权重向量对应一个分类。每一个向量包含13个权重值，通过与13维的葡萄酒数

据集中的特征数据相乘来计算模型的净输入：

$$z = w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}$$

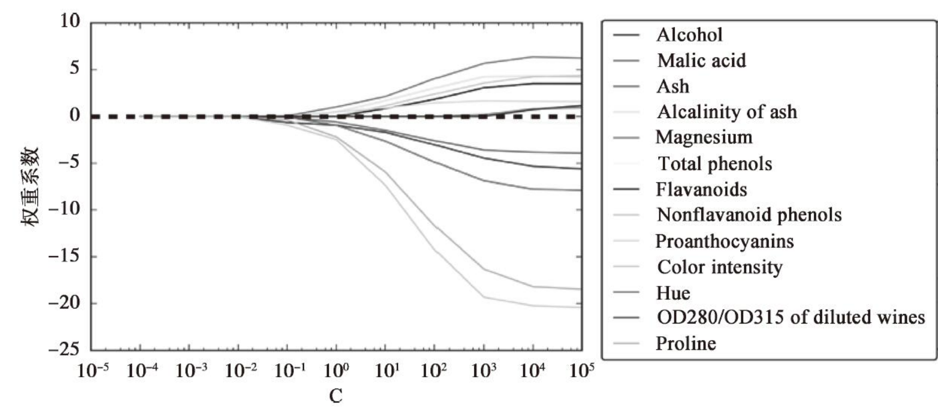
我们注意到，权重向量是稀疏的，这意味着只有少数几个非零项。在L1正则化特征选择的作用下，我们训练了一个模型，该模型对数据集上可能存在的不相关特征是鲁棒的。

最后，我们来绘制一下正则化效果的图，它展示了将权重系数（正则化参数）应用于多个特征上时所产生的不同的正则化效果：

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> colors = ['blue', 'green', 'red', 'cyan',
...           'magenta', 'yellow', 'black',
...           'pink', 'lightgreen', 'lightblue',
...           'gray', 'indigo', 'orange']
>>> weights, params = [], []
>>> for c in np.arange(-4, 6):
...     lr = LogisticRegression(penalty='l1',
...                             C=10**c,
...                             random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> for column, color in zip(range(weights.shape[1]), colors):
...     plt.plot(params, weights[:, column],
...              label=df_wine.columns[column+1],
...              color=color)
>>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>>> plt.xlim([10**(-5), 10**5])
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.xscale('log')
>>> plt.legend(loc='upper left')
>>> ax.legend(loc='upper center',
...           bbox_to_anchor=(1.38, 1.03),
...           ncol=1, fancybox=True)
>>> plt.show()
```

通过下图，我们能对L1正则化有个更深入的认识。可以看到，在强的

正则化参数（C<0.1）作用下，罚项使得所有的特征权重都趋近于0，这里，C是正则化参数的倒数。



4.5.2　序列特征选择算法

另外一种降低模型复杂度从而解决过拟合问题的方法是通过特征选择进行降维（dimensionality reduction），该方法对未经正则化处理的模型特别有效。降维技术主要分为两个大类：特征选择和特征提取。通过特征选择，我们可以选出原始特征的一个子集。而在特征提取中，通过对现有的特征信息进行推演，构造出一个新的特征子空间。在本节，我们将着眼于一些经典的特征选择算法。第5章会介绍几种特征抽取技术，以将数据集压缩到低维的特征子空间。

序列特征选择算法系一种贪婪搜索算法，用于将原始的d维特征空间压缩到一个k维特征子空间，其中k＜d。使用特征选择算法出于以下考虑：能够剔除不相关特征或噪声，自动选出与问题最相关的特征子集，从而提高计算效率或是降低模型的泛化误差。这在模型不支持正则化时尤为有效。一个经典的序列特征选择算法是序列后向选择算法（Sequential Backward Selection，SBS），其目的是在分类性能衰减最小的约束下，降低原始特征空间上的数据维度，以提高计算效率。在某些情况下，SBS甚至可以在模型面临过拟合问题时提高模型的预测能力。

与穷举搜索算法相比，贪心算法可以在组合搜索的各阶段找到一个局部最优选择，进而得到待解决问题的次优解。不过在实际应用中，由于巨大的计算成本，往往使得穷举搜索算法方法不可行，而贪心算法则可以找到不那么复杂，且计算效率更高的解决方案。

SBS算法背后的理念非常简单：SBS依次从特征集合中删除某些特征，

直到新的特征子空间包含指定数量的特征。为了确定每一步中所需删除的特征，我们定义一个需要最小化的标准衡量函数J。该函数的计算准则是：比较判定分类器的性能在删除某个特定特征前后的差异。由此，每一步中待删除的特征，就是那些能够使得标准衡量函数值尽可能大的特征，或者更直观地说：每一步中特征被删除后，所引起的模型性能损失最小。基于上述对SBS的定义，我们可以将算法总结为四个简单的步骤：

1）设$k=d$进行算法初始化，其中d是特征空间X_d的维度。

2）定义x^-为满足标准$x^-=\arg\max J(X_k-x)$最大化的特征，其中$x\in X_k$。

3）将特征x^-从特征集中删除：$X_{k-1}=X_k-x^-$，$k=k-1$。

4）如果k等于目标特征数量，算法终止，否则跳转到第2步。

如果想更多了解序列特征选择算法的相关问题，请参阅文献：Comparative Study of Techniques for Large Scale Feature Selection,F.Ferri,P.Pudil,M.Hatef,and J.Kittler.Comparative study of techniques for large-scale feature selection.Pattern Recognition in Practice IV,pages 403——413，1994。

遗憾的是，scikit-learn中并没有实现SBS算法。不过它相当简单，我们可以使用Python来实现：

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.cross_validation import train_test_split
from sklearn.metrics import accuracy_score
class SBS():
    def __init__(self, estimator, k_features,
        scoring=accuracy_score,
        test_size=0.25, random_state=1):
        self.scoring = scoring
```

```
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state

    def fit(self, X, y):
        X_train, X_test, y_train, y_test = \
                train_test_split(X, y, test_size=self.test_size,
                                 random_state=self.random_state)

        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                 X_test, y_test, self.indices_)
        self.scores_ = [score]

        while dim > self.k_features:
            scores = []
            subsets = []

            for p in combinations(self.indices_, r=dim-1):
                score = self._calc_score(X_train, y_train,
                                         X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]

        return self

    def transform(self, X):
        return X[:, self.indices_]

    def _calc_score(self, X_train, y_train,
                    X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score
```

在前面的实现中，我们使用参数k_features来指定需返回的特征数量。默认情况下，我们使用scikit-learn中的accuracy_score去衡量分类器的模型和评估器在特征空间上的性能。在fit方法的while循环中，通过itertools.combination函数创建的特征子集循环地进行评估和删减，直到

特征子集达到预期维度。在每次迭代中，基于内部测试数据集X_test创建的self.scors_列表存储了最优特征子集的准确度分值。后续我们将使用这些分值来对结果做出评价。最终特征子集的列标被赋值给self.indices_，我们可以通过transform方法返回由选定特征列构成的一个新数组。请注意，我们没有在fit方法中明确地计算评价标准，而只是简单地删除了那些没有包含在最优特征子集中的特征。

现在，来看一下我们实现的SBS应用于scikit-learn中KNN分类器的效果：

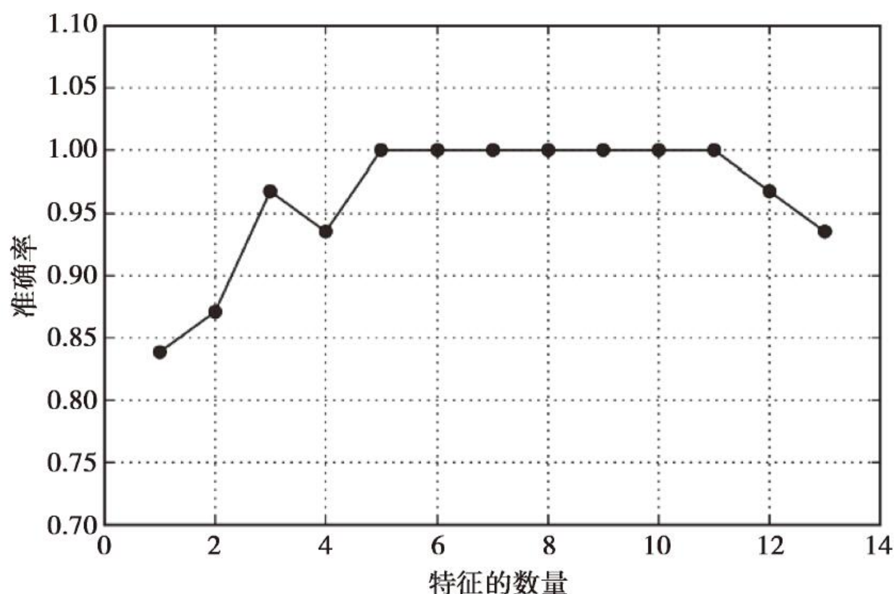
```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> import matplotlib.pyplot as plt
>>> knn = KNeighborsClassifier(n_neighbors=2)
>>> sbs = SBS(knn, k_features=1)
>>> sbs.fit(X_train_std, y_train)
```

在SBS的实现中，已经在fit函数内将数据集划分为测试数据集和训练数据集，我们依旧将训练数据集X_train输入到算法中。SBS的fit方法将建立一个新的训练子集用于测试（验证）和训练，这也是为什么这里的测试数据集被称为验证数据集的原因。这也是一种避免原始测试数据集变成训练数据集的必要方法。

我们的SBS算法在每一步中都存储了最优特征子集的分值，下面进入代码实现中更为精彩的部分：绘制出KNN分类器的分类准确率，准确率数值是在验证数据集上计算得出的。代码如下：

```
>>> k_feat = [len(k) for k in sbs.subsets_]
>>> plt.plot(k_feat, sbs.scores_, marker='o')
>>> plt.ylim([0.7, 1.1])
>>> plt.ylabel('Accuracy')
>>> plt.xlabel('Number of features')
>>> plt.grid()
>>> plt.show()
```

通过下图可以看到：当我们减少了特征的数量后，KNN分类器在验证数据集上的准确率提高了，这可能归因于在第3章中介绍KNN算法时讨论过的“维度灾难”。此外，图中还显示，当k = {5，6，7，8，9，10}时，算法可以达到百分之百的准确率。



为了满足一下自己的好奇心，现在让我们看一下是哪五个特征在验证数据集上有如此好的表现：

```
>>> k5 = list(sbs.subsets_[8])
>>> print(df_wine.columns[1:][k5])
Index(['Alcohol', 'Malic acid', 'Alcalinity of ash', 'Hue',
'Proline'], dtype='object')
```

使用上述代码，我们从sb.subsets_的第9列中获取了五个特征子集的列标，并通过以pandas的DataFrame格式存储的葡萄酒数据对应的索引中提取到了相应的特征名称。

下面我们验证一下KNN分类器在原始测试集上的表现：

```
>>> knn.fit(X_train_std, y_train)
>>> print('Training accuracy:', knn.score(X_train_std, y_train))
Training accuracy: 0.983870967742
>>> print('Test accuracy:', knn.score(X_test_std, y_test))
Test accuracy: 0.944444444444
```

在上述代码中，我们在训练数据集上使用了所有的特征并得到了大约为98.4%的准确率。不过，在测试数据集上的准确率稍低（约为94.4%），此指标暗示模型稍有过拟合。现在让我们在选定的五个特征集上看一下KNN的性能：

```
>>> knn.fit(X_train_std[:, k5], y_train)
>>> print('Training accuracy:',
...        knn.score(X_train_std[:, k5], y_train))
Training accuracy: 0.959677419355
>>> print('Test accuracy:',
...        knn.score(X_test_std[:, k5], y_test))
Test accuracy: 0.962962962963
```

当特征数量不及葡萄酒数据集原始特征数量的一半时，在测试数据集上的预测准确率提高了两个百分点。同时，通过测试数据集（约为96.3%）和训练数据集（约为96.0%）上准确率之间的微小差别可以发现，过拟合现象得以缓解。


scikit-learn中的特征选择算法

scikti-learn提供了许多其他的特征选择算法，包括：基于特征权重的

递归后向消除算法（recursive backward elimination based on feature weights）、基于特征重要性的特征选择树方法（tree-based methods to select features by importance），以及单变量统计测试（univariate statistical tests）。对特征选择方法进行更深入的讨论已经超出了本书的范围，对这些算法的总结性描述及其示例请参见链接：http://scikit-learn.org/stable/modules/feature_selection.html。

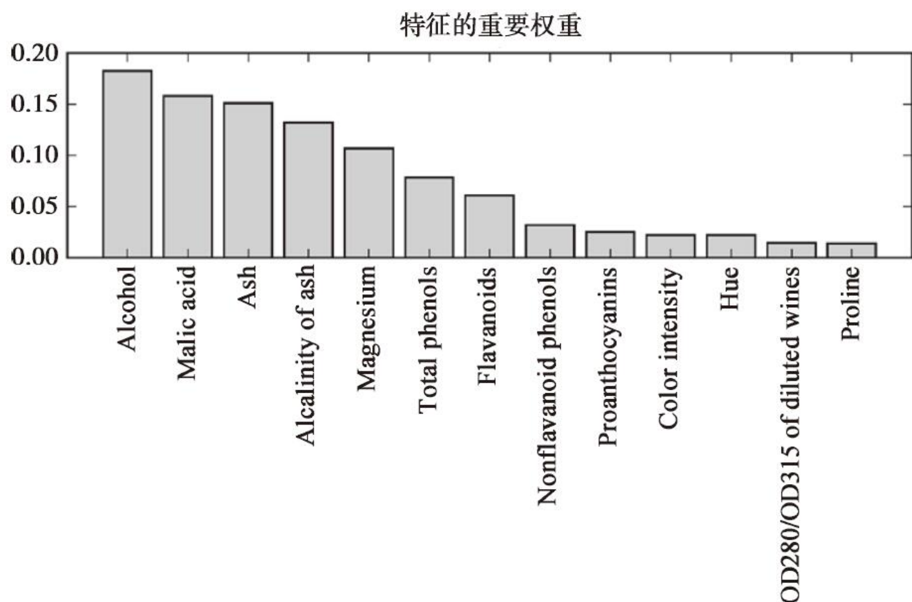
4.6 通过随机森林判定特征的重要性

在前面几节，我们学习了如何在逻辑斯谛回归上通过L1正则化将不相关特征剔除，并且学习了用于特征选择的SBS算法。另一种从数据集中选择相关特征的有效方法是使用随机森林，也就是在第3章中介绍的集成技术。我们可以通过森林中所有决策树得到的平均不纯度衰减来度量特征的重要性，而不必考虑数据是否线性可分。更加方便的是：scikit-learn中实现的随机森林已经为我们收集好了关于特征重要程度的信息，在拟合了RandomForestClassifier后，可以通过feature_importances_得到这些内容。执行以下代码，可在葡萄酒数据集上训练10000棵树，并且分别根据其重要程度对13个特征给出重要性等级。请记住（在第3章中曾讨论过的）：无需对基于树的模型做标准化或归一化处理。代码如下：

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> feat_labels = df_wine.columns[1:]
>>> forest = RandomForestClassifier(n_estimators=10000,
...                                 random_state=0,
...                                 n_jobs=-1)
```

```
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[f],
...                             importances[indices[f]]))
1) Alcohol                        0.182508
2) Malic acid                     0.158574
3) Ash                            0.150954
4) Alcalinity of ash              0.131983
5) Magnesium                      0.106564
6) Total phenols                  0.078249
7) Flavanoids                     0.060717
8) Nonflavanoid phenols           0.032039
9) Proanthocyanins                0.025385
10) Color intensity               0.022369
11) Hue                           0.022070
12) OD280/OD315 of diluted wines  0.014655
13) Proline                       0.013933
>>> plt.title('Feature Importances')
>>> plt.bar(range(X_train.shape[1]),
...         importances[indices],
...         color='lightblue',
...         align='center')
>>> plt.xticks(range(X_train.shape[1]),
...            feat_labels, rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()
```

执行上述代码后，根据特征在葡萄酒数据集中的相对重要性，绘制出根据特征重要性排序的图，请注意，这些特征重要性经过归一化处理，其和为1.0。



由上图我们可以得到结论：基于10000棵决策树平均不纯度衰减的计算，数据集上最具判别效果的特征是“alcohol”。有趣的是，上图中排名靠前的四个特征中有三个也出现在前面章节中使用SBS算法所选的五个特征中。不过，就可解释性而言，随机森林有一个重要的特性值得一提。例如：如果两个或者更多个特征是高度相关的，一个特征的相关特征未被完全包含进来，那么此特征可能会得到一个较高的评分。另一方面，如果我们的关注点仅在模型的预测性能上，而不关心对特征重要性解释，那么就不必纠缠于上述问题。在总结本小节关于特征重要性和随机森林的内容之前，需要提醒一下：在完成对模型的拟合后，scikit-learn还实现了一个transofrm方法，可以在用户设定阈值的基础上进行特征选择，这在我们将RandomForestClassifier作为特征选择器，并将其作为scikit-learn实现分析流程环节中的一个步骤时尤为有效，它使得我们可以通过同一个预估器来连接不同的预处理步骤，我们将在第6章中对这些相关内容进行探讨。例

如，将阈值设为0.15，我们可以使用下列代码将数据集压缩到三个最重要的特征：Alcohol、Malic acid和Ash：

```
>>> X_selected = forest.transform(X_train, threshold=0.15)
>>> X_selected.shape
(124, 3)
```

本章小结

本章的开始着眼于正确处理缺失数据的有用技术。在我们将数据导入到机器学习算法之前，应保证已对类别变量进行了正确的编码，我们还分别讨论了如何将有序特征和标称特征的值映射为整数的方法。

此外，我们还简要地讨论了L1正则化，它可以通过降低模型的复杂度来帮助我们避免过拟合。作为另外一种剔除不相关特征的方法，我们使用了序列特征选择算法从数据集中选择有意义的特征。

在下一章中，读者将学到另外一种降维的有效方法：特征提取。它使得我们可以将特征压缩到一个低维空间，而不是像特征选择那样完全剔除不相关特征。

第5章　通过降维压缩数据

在第4章中，我们介绍了几种通过不同的特征选择技术对数据集进行降维的方法。另一种常用于降维的特征选择方法就是特征抽取。在本章中，读者将学习三种可以帮助我们归纳总结数据集内所蕴含信息的技术，它们都可以将原始数据集变换到一个维度更低的新的特征子空间。数据压缩也是机器学习领域中的一个重要内容，随着现代技术的发展，将会产生越来越多的数据，数据压缩技术可以帮助我们对数据进行存储和分析。本章将涵盖如下主题：

·无监督数据压缩——主成分分析（Principal Component Analysis，PCA）

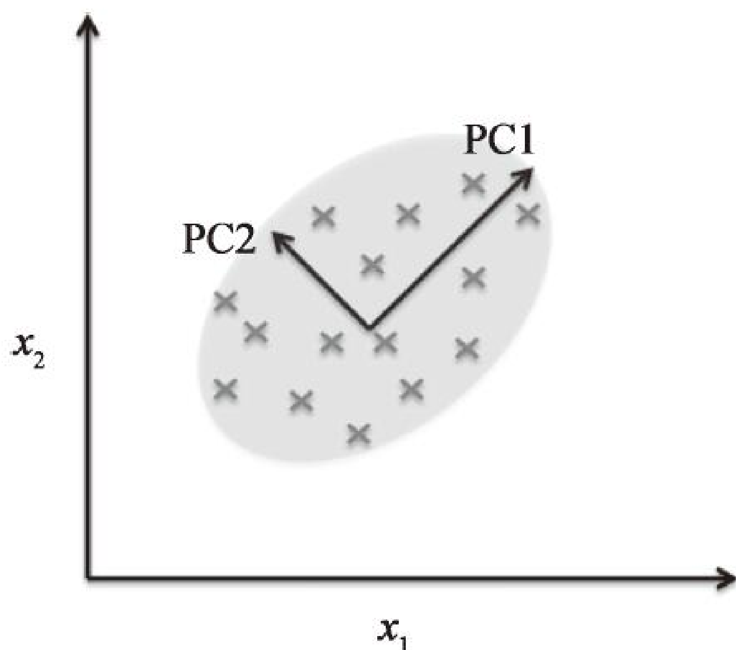
·基于类别可分最大化的监督降维技术——线性判别分析（Linear Discriminant Analysis，LDA）

·通过核主成分分析（kernel principal component analysis）进行非线性降维

5.1　无监督数据降维技术——主成分分析

与特征选择类似，我们可以使用特征抽取来减少数据集中特征的数量。不过，当使用序列后向选择等特征选择算法时，能够保持数据的原始特征，而特征抽取算法则会将数据转换或者映射到一个新的特征空间。基于降维在数据预处理领域的含义，特征抽取可以理解为：在尽可能多地保持相关信息的情况下，对数据进行压缩的一种方法。特征抽取通常用于提高计算效率，同样也可以帮助我们降低“维度灾难”——尤其当模型不适于正则化处理时。

主成分分析（principal component analysis，PCA）是一种广泛应用于不同领域的无监督线性数据转换技术，其突出作用是降维。PCA的其他常用领域包括：股票交易市场数据的探索性分析和信号去噪，以及生物信息学领域的基因组和基因表达水平数据分析等。PCA可以帮助我们基于特征之间的关系识别出数据内在的模式。简而言之，PCA的目标是在高维数据中找到最大方差的方向，并将数据映射到一个维度不大于原始数据的新的子空间上。如下图所示，以新特征的坐标是相互正交的为约束条件，新的子空间上正交的坐标轴（主成分）可被解释为方差最大的方向。在此，x_1和x_2为原始特征的坐标轴，而PC1和PC2即为主成分。



如果使用PCA降维，我们将构建一个d×k维的转换矩阵W，这样就可以将一个样本向量x映射到一个新的k维特征子空间上去，此空间的维度小于原始的d维特征空间：

$$\boldsymbol{x}=[x_1, x_2, \cdots, x_d],\ \boldsymbol{x}\in\mathbb{R}^d$$
$$\downarrow \boldsymbol{xW},\ \boldsymbol{W}\in\mathbb{R}^{d\times k}$$
$$\boldsymbol{z}=[z_1, z_2, \cdots, z_k],\ \boldsymbol{z}\in\mathbb{R}^k$$

完成从原始d维数据到新的k维子空间（一般情况下k＜＜d）的转换后，第一主成分的方差应该是最大的，由于各主成分之间是不相关的（正交的），后续各主成分也具备尽可能大的方差。需注意的是，主成分的方向对数据值的范围高度敏感，如果特征的值使用不同的度量标准，我们需要先对特征进行标准化处理，以让各特征具有相同的重要性。

在详细讨论使用PCA算法降维之前，我们先通过以下几个步骤来概括一下算法的流程：

1）对原始d维数据集做标准化处理。

2）构造样本的协方差矩阵。

3）计算协方差矩阵的特征值和相应的特征向量。

4）选择与前k个最大特征值对应的特征向量，其中k为新特征空间的维度（$k \leq d$）。

5）通过前k个特征向量构建映射矩阵W。

6）通过映射矩阵W将d维的输入数据集X转换到新的k维特征子空间。

5.1.1 总体方差与贡献方差

在本小节中，我们将学习主成分分析算法的前四个步骤：数据标准化、构造协方差矩阵、获得协方差矩阵的特征值和特征向量，以及按降序排列特征值所对应的特征向量：

首先，我们加载第4章已经使用过的葡萄酒数据集：

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/wine/wine.data', header=None)
```

接下来，我们将葡萄酒数据集划分为训练集和测试集——分别占数据集的70%和30%，并使用单位方差将其标准化。

```
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...                 train_test_split(X, y,
...                 test_size=0.3, random_state=0)
>>> sc = StandardScaler()

>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.fit_transform(X_test)
```

通过上述代码完成数据的预处理后，我们进入第二步：构造协方差矩阵。此d×d维协方差矩阵是沿主对角线对称的，其中d为数据集的维度，此矩阵成对地存储了不同特征之间的协方差。例如，对群体进行描述的两个特征x_j和x_k可通过如下公式计算它们之间的协方差：

$$\sigma_{jk}=\frac{1}{n}\sum_{i=1}^{n}(x_j^{(i)}-\mu_j)(x_k^{(i)}-\mu_k)$$

在此，μ_j和μ_k分别为特征j和k的均值。请注意，如果我们对数据集做了标准化处理，样本的均值将为零。两个特征之间的协方差如果为正，说明它们会同时增减，而一个负的协方差值则表示两个特征会朝相反的方向变动。例如，一个包含三个特征的协方差矩阵可记为（注意：此处Σ代表希腊字母sigma，在此请勿与求和符号混为一谈）：

$$\Sigma=\begin{bmatrix}\sigma_1^2 & \sigma_{12} & \sigma_{13}\\ \sigma_{21} & \sigma_2^2 & \sigma_{23}\\ \sigma_{31} & \sigma_{32} & \sigma_3^2\end{bmatrix}$$

协方差矩阵的特征向量代表主成分（最大方差方向），而对应的特征值大小就决定了特征向量的重要性。就葡萄酒数据集来说，我们可以得到13×13维协方差矩阵的13个特征向量及其对应的特征值。

现在，我们来计算协方差矩阵的特征对。通过线性代数或微积分的相关知识我们知道，特征值V需满足如下条件：

$$\Sigma v = \lambda v$$

此处的特征值是λ一个标量。由于手工计算特征向量和特征值从某种程度上来说是一项烦琐且复杂的工作，我们将使用NumPy中的linalg.eig函数来计算葡萄酒数据集协方差矩阵的特征对：

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nEigenvalues \n%s' % eigen_vals)
Eigenvalues
[ 4.8923083   2.46635032  1.42809973  1.01233462  0.84906459
0.60181514
0.52251546  0.08414846  0.33051429  0.29595018  0.16831254  0.21432212
0.2399553 ]
```

应用numpy.cov函数，我们计算得到了经标准化处理的训练数据集的协方差矩阵。使用linalg.eig函数，通过特征分解，得到一个包含13个特征值的向量（eigen_vals）及其对应的特征值，特征向量以列的方式存储于一个13×13维的矩阵中（eigen_vecs）。

因为要将数据集压缩到一个新的特征子空间上来实现数据降维，所以我们只选择那些包含最多信息（方差最大）的特征向量（主成分）组成子集。由于特征值的大小决定了特征向量的重要性，因此需要将特征值按降序排列，我们感兴趣的是排序在前k个的特征值所对应的特征向量。在整理包含信息量最大的前k个特征向量前，我们先绘制特征值的方差贡献率（variance explained ratios）图像。

特征值λ_j的方差贡献率是指，特征值λ_j与所有特征值和的比值：

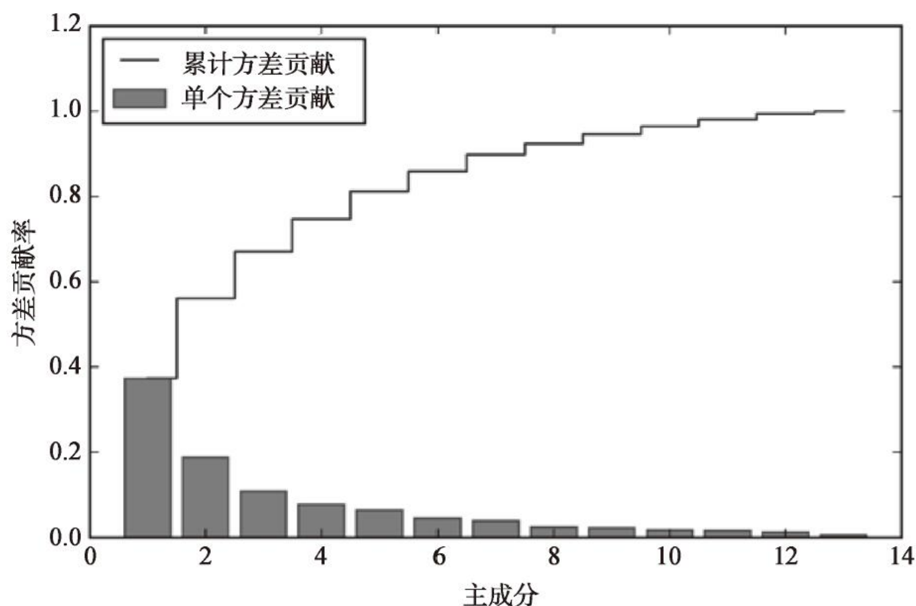
$$\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$$

使用NumPy的cumsum函数，我们可以计算出累计方差，其图像可通过matplotlib的step函数绘制：

```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...            sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)

>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, alpha=0.5, align='center',
...         label='individual explained variance')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...         label='cumulative explained variance')
>>> plt.ylabel('Explained variance ratio')
>>> plt.xlabel('Principal components')
>>> plt.legend(loc='best')
>>> plt.show()
```

由下图可以看到，第一主成分占方差总和的40%左右；此外，还可以看出前两个主成分占总体方差的近60%：



虽然方差贡献率图像可以让我们联想到第4章中通过随机森林计算出的关于特征的重要程度，但我们应注意：PCA是一种无监督方法，这意味着我们可以忽略类标信息。相对而言，随机森林通过类标信息来计算节点

的不纯度，而方差度量的是特征值在轴线上的分布。

5.1.2 特征转换

在将协方差矩阵分解为特征对后，我们继续执行PCA方法的最后三个步骤，将葡萄酒数据集中的信息转换到新的主成分轴上。在本小节中，我们将对特征值按降序进行排列，并通过挑选出对应的特征向量构造出映射矩阵，然后使用映射矩阵将数据转换到低维的子空间上。

首先，按特征值的降序排列特征对：

```
>>> eigen_pairs =[(np.abs(eigen_vals[i]),eigen_vecs[:,i])
...                for i inrange(len(eigen_vals))]
>>> eigen_pairs.sort(reverse=True)
```

接下来，我们选取两个对应的特征值最大的特征向量，这两个值之和占据了数据集总体方差的60%。请注意，出于演示的需要，我们只选择了两个特征向量，因为在本小节中我们将以二维散点图的方式绘制相关图像。在实际应用中，确定主成分的数量时，我们需要根据实际情况在计算效率与分类器性能之间做出权衡。

```
>>> w= np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                eigen_pairs[1][1][:, np.newaxis]))
>>> print('Matrix W:\n',w)
Matrix W:
[[ 0.14669811  0.50417079]
[-0.24224554  0.24216889]
[-0.02993442  0.28698484]
[-0.25519002 -0.06468718]
[ 0.12079772  0.22995385]
[ 0.38934455  0.09363991]
[ 0.42326486  0.01088622]
[-0.30634956  0.01870216]
[ 0.30572219  0.03040352]
[-0.09869191  0.54527081]
[ 0.30032535 -0.27924322]
[ 0.36821154 -0.174365  ]
[ 0.29259713  0.36315461]]
```

执行上述代码，通过选取的两个特征向量我们得到了一个13×2维的映射矩阵W。通过映射矩阵，我们可以将一个样本x（以13维的行向量表示）转换到PCA的子空间上得到x'，样本转换为一个包含两个新特征的二维向量。

$$x'=xW$$

```
>>> X_train_std[0].dot(w)
array([ 2.59891628,  0.00484089])
```

类似地，通过计算矩阵的点积，我们可以将整个124×13维的训练数据集转换到包含两个主成分的子空间上：

$$X'=XW$$

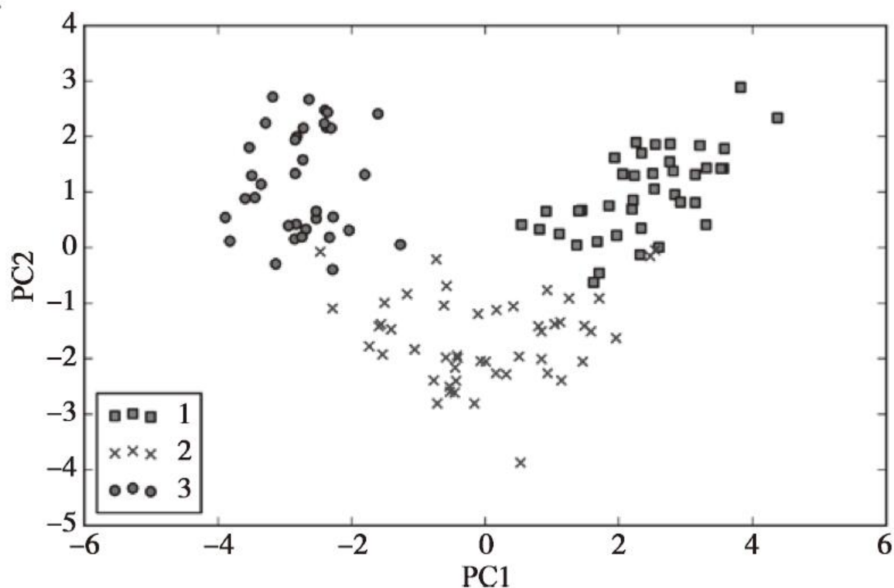
```
>>> X_train_pca = X_train_std.dot(w)
```

最后，转换后的葡萄酒数据集将以124×2维矩阵的方式存储，我们以二维散点图的方式来对其进行可视化展示：

```
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=l, marker=m)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')

>>> plt.legend(loc='lower left')
>>> plt.show()
```

在下图中可以看到，相较于第二主成分（y轴），数据更多地沿着x轴（第一主成分）方向分布，这与我们在上一小节中绘制的方差贡献率图是一致的。而且，可以直观地看到，线性分类器能够很好地对其进行划分：



为了演示散点图，我们使用了类标信息，但请注意PCA是无监督方法，无需类标信息。

5.1.3　使用scikit-learn进行主成分分析

通过上一小节中详尽的介绍，我们了解了PCA内部的工作原理，接下来讨论如何使用scikit-learn中提供的PCA类。PCA也是scikit-learn中的一个数据转换类，在使用相同的模型参数对训练数据和测试数据进行转换之前，我们首先要使用训练数据对模型进行拟合。下面，我们使用scikit-learn中的PCA对葡萄酒数据集做预处理，然后使用逻辑斯谛回归对转换后的数据进行分类，最后用第2章中定义的plot_decision_region函数对决策区域进行可视化展示。

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface

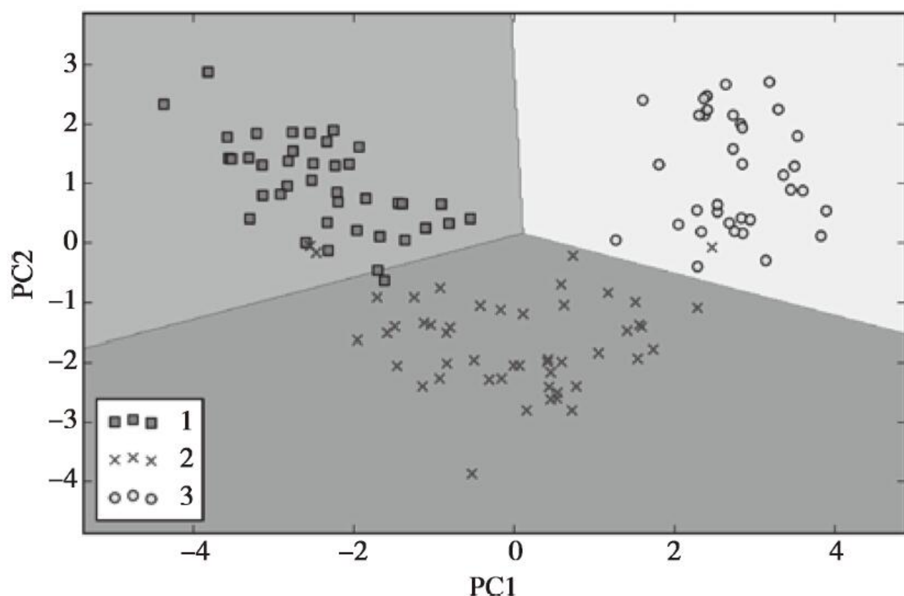
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
```

```
    plt.ylim(xx2.min(), xx2.max())

    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)

>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression()
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

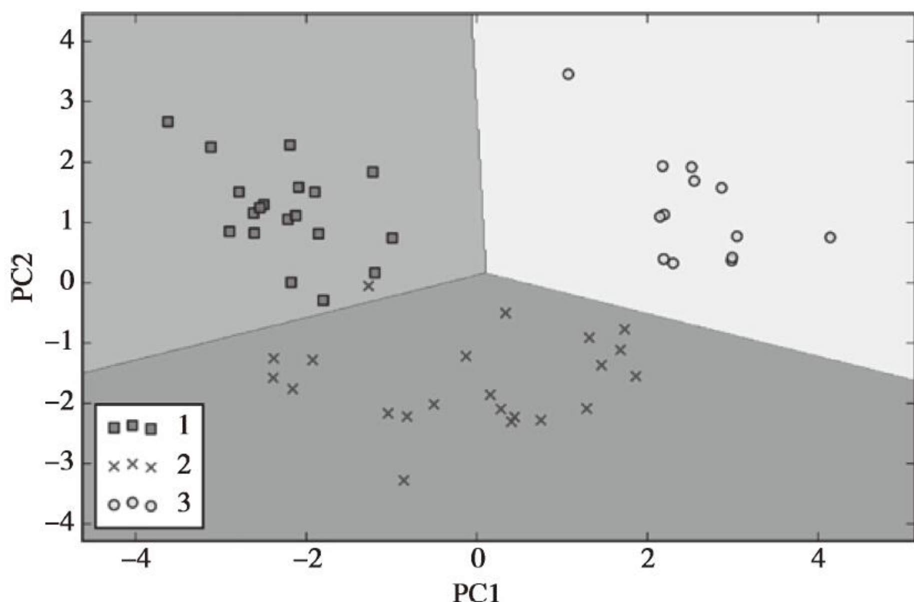
执行上述代码后，我们可以看到将训练数据转换到两个主成分轴后生成的决策区域。



如果比较scikit-learn中提供的PCA类与我们自己实现的PCA类的分析结果，可以发现：上图可以看作是我们此前自己完成PCA方法所得到沿y轴进行旋转后的结果。出现此差异的原因，不是两种方法在实现中出现了什么错误，而在于特征分析方法：特征向量可以为正或者为负。这不是重点，因为有需要时我们可以在数据上乘以-1来实现图像的镜像。注意：特征向量通常会缩放到单位长度1。为了保证整个分析过程的完整性，我们绘制一下逻辑斯谛回归在转换后的测试数据上所得到的决策区域，看其是否能很好地将各类分开：

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

执行上述代码，我们绘制出了测试集的决策区域，可以看到：逻辑斯谛回归在这个小的二维特征子空间上表现优异，只误判了一个样本。



如果我们对不同主成分的方差贡献率感兴趣，可以将PCA类中的n_componets参数设置为None，由此，可以保留所有的主成分，并且可以通过explained_variance_ratio_属性得到相应的方差贡献率：

```
>>> pca = PCA(n_components=None)
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> pca.explained_variance_ratio_
array([ 0.37329648,  0.18818926,  0.10896791,  0.07724389,
0.06478595,
0.04592014,  0.03986936,  0.02521914,  0.02258181,  0.01830924,
0.01635336,  0.01284271,  0.00642076])
```

请注意：在初始化PCA类时，如果我们将n_components设置为None，那么它将按照方差贡献率递减顺序返回所有的主成分，而不是进行降维操作。

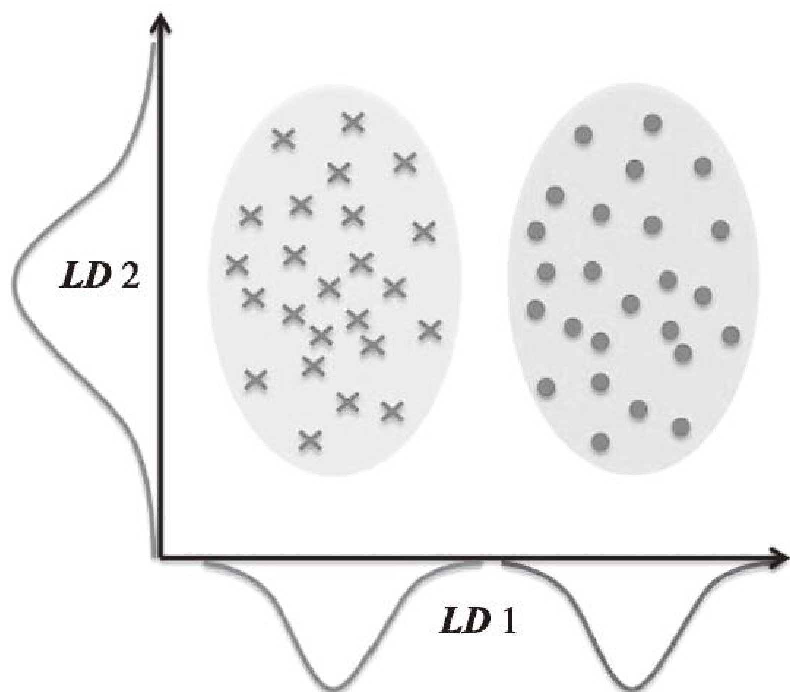
5.2　通过线性判别分析压缩无监督数据

线性判别分析（Linear Discriminant Analysis，LDA）是一种可作为特征抽取的技术，它可以提高数据分析过程中的计算效率，同时，对于不适用于正则化的模型，它可以降低因维度灾难带来的过拟合。

LDA的基本概念与PCA非常相似，PCA试图在数据集中找到方差最大的正交的主成分分量的轴，而LDA的目标是发现可以最优化分类的特征子空间。LDA与PCA都是可用于降低数据集维度的线性转换技巧。其中，PCA是无监督算法，而LDA是监督算法。因此，我们可以这样直观地认为：与PCA相比，LDA是一种更优越的用于分类的特征提取技术。但是A.M.Martinez提出：在图像识别任务中的某些情况下，如每个类别中只有少量样本，使用PCA作为预处理工具的分类结果更佳[1]。

LDA有时也被称作Fisher's LDA，Ronald A.Fisher于1936年针对二类别分类问题对Fisher线性判别（Fisher's Linear Discriminant）做了最初的形式化[2]。1948年，基于类别方差相等和类内样本呈标准正态分布的假设，Radhakrishna Rao将Fisher's LDA泛化到了多类别分类问题上，即我们现在所说的LDA[3]。

下图解释了二类别分类中LDA的概念。类别1、类别2中的样本分别用叉号和原点来表示：



如上图所示，在x轴方向（LD1），通过线性判定，可以很好地将呈正态分布的两个类分开。虽然沿y轴（LD2）方向的线性判定保持了数据集的较大方差，但是沿此方向无法提供关于类别区分的任何信息，因此它不是一个好的线性判定。

一个关于LDA的假设就是数据呈正态分布。此外，我们还假定各类别中数据具有相同的协方差矩阵，且样本的特征从统计上来讲是相互独立的。不过，即使一个或多个假设没有满足，LDA仍旧可以很好地完成降维工作[4]。

在进入下一节详细讨论LDA的原理之前，我们先来总结一下LDA方法的关键步骤：

1）对d维数据集进行标准化处理（d为特征的数量）。

2）对于每一类别，计算d维的均值向量。

3）构造类间的散布矩阵S_B以及类内的散布矩阵S_W。

4）计算矩阵$S_W^{-1}S_B$的特征值及对应的特征向量。

5）选取前k个特征值所对应的特征向量，构造一个d×k维的转换矩阵W，其中特征向量以列的形式排列。

6）使用转换矩阵W将样本映射到新的特征子空间上。

特征呈正态分布且特征间相互独立是我们使用LDA时所做的假设。同时，LDA算法假定各个类别的协方差矩阵是一致的。然而，即使我们违背了上述假设，LDA算法仍旧能很好地完成数据降维及分类任务（R.O.Duda,P.E.Hart，and D.G.Stork.Pattern Classification.2nd.Edition.New York,2001）。

[1] A.M.Martinez and A.C.Kak.PCA Versus LDA.Pattern Analysis and Machine Intelligence，IEEE Transactions on，23（2）：228-233,2001.

[2] R.A. Fisher.The Use of Multiple Measuremengts in Taxonomic Problems.Annals of Eugenics，7(2):179-188，1936.

[3] C. R. Rao. The Utilization of Multiple Measurements in Problems of Biological Classification. Journal of the Royal Statistical Society. Series B（Methodological），10(2):159—203，1948.

[4] R. O. Duda, P. E. Hart, and D. G. Stork. Pattern Classification. 2nd.

Edition. New York, 2001.

5.2.1 计算散布矩阵

本节伊始，在讲解PCA时就对葡萄酒数据集做了标准化处理，因此我们将跳过第一步直接计算均值向量，计算中，我们将分别构建类内散布矩阵和类间散布矩阵。均值向量m_i存储了类别i中样本的特征均值μ_m：

$$\boldsymbol{m}_i = \frac{1}{n_i}\sum_{x\in D_i}^{c} \boldsymbol{x}_m$$

葡萄酒数据集的三个类别对应三个均值向量：

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix} \quad i \in \{1,2,3\}$$

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(
...                X_train_std[y_train==label], axis=0))
...     print('MV %s: %s\n' %(label, mean_vecs[label-1]))
MV 1: [ 0.9259 -0.3091  0.2592 -0.7989  0.3039  0.9608  1.0515 -0.6306
0.5354
  0.2209  0.4855  0.798   1.2017]

MV 2: [-0.8727 -0.3854 -0.4437  0.2481 -0.2409 -0.1059  0.0187 -0.0164
0.1095
 -0.8796  0.4392  0.2776 -0.7016]

MV 3: [ 0.1637  0.8929  0.3249  0.5658 -0.01   -0.9499 -1.228   0.7436
-0.7652
  0.979  -1.1698 -1.3007 -0.3912]
```

通过均值向量，我们来计算一下类内散布矩阵S_W：

$$\boldsymbol{S}_W = \sum_{i=1}^{c} \boldsymbol{S}_i$$

这可以通过累加各类别i的散布矩阵S_i来计算：

$$S_i=\sum_{x\in D_i}^{c}(x-m_i)(x-m_i)^{\mathrm{T}}$$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1,4), mean_vecs):
...     class_scatter = np.zeros((d, d))
...     for row in X[y == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row-mv).dot((row-mv).T)
...     S_W += class_scatter
>>> print('Within-class scatter matrix: %sx%s'
...        % (S_W.shape[0], S_W.shape[1]))
Within-class scatter matrix: 13x13
```

此前我们对散布矩阵进行计算时，曾假设训练集的类标是均匀分布的。但是，通过打印类标的数量，可以看到在此并未遵循此假设：

```
>>> print('Class label distribution: %s'
...       % np.bincount(y_train)[1:])
Class label distribution: [40 49 35]
```

因此，在我们通过累加方式计算散布矩阵S_W前，需要对各类别的散布矩阵S_i做缩放处理。当我们用各类别单独的散布矩阵除以此类别内样本数量N_i时，可以发现计算散布矩阵的方式与计算协方差矩阵$\sum_i$的方式是一致的。协方差矩阵可以看作是归一化的散布矩阵：

$$\sum_i=\frac{1}{N_i}S_w=\frac{1}{N_i}\sum_{x\in D_i}^{c}(x-m_i)(x-m_i)^{\mathrm{T}}$$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.cov(X_train_std[y_train==label].T)
...     S_W += class_scatter
>>> print('Scaled within-class scatter matrix: %sx%s'
...        % (S_W.shape[0], S_W.shape[1]))
Scaled within-class scatter matrix: 13x13
```

在完成类内散布矩阵（或协方差矩阵）的计算后，我们进入下一步骤，计算类间散布矩阵S_B：

$$\boldsymbol{S}_B=\sum_{i=1}^{c}N_i(\boldsymbol{m}_i-\boldsymbol{m})(\boldsymbol{m}_i-\boldsymbol{m})^{\mathrm{T}}$$

其中，m为全局均值，它在计算时用到了所有类别中的全部样本：

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> d = 13 # number of features
>>> S_B = np.zeros((d, d))
>>> for i,mean_vec in enumerate(mean_vecs):
...     n = X[y==i+1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1)
...     mean_overall = mean_overall.reshape(d, 1)
    S_B += n * (mean_vec - mean_overall).dot(
...                 (mean_vec - mean_overall).T)
print('Between-class scatter matrix: %sx%s'
...     % (S_B.shape[0], S_B.shape[1]))
Between-class scatter matrix: 13x13
```

5.2.2　在新特征子空间上选取线性判别算法

LDA余下的步骤与PCA的步骤相似。不过，这里我们不对协方差矩阵做特征分解，而是求解矩阵$S_W^{-1}S_B$的广义特征值：

```
>>>eigen_vals, eigen_vecs =\
...np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

在求得了特征对之后，我们按照降序对特征值进行排序：

```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...                for i in range(len(eigen_vals))]
>>> eigen_pairs = sorted(eigen_pairs,
...                 key=lambda k: k[0], reverse=True)
>>> print('Eigenvalues in decreasing order:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])


Eigenvalues in decreasing order:

643.015384346
225.086981854
1.37146633984e-13
5.68434188608e-14
4.16877714935e-14
4.16877714935e-14
3.76733516161e-14
3.7544790902e-14
3.7544790902e-14
2.30295239559e-14
2.30295239559e-14
1.9101018959e-14
3.86601693797e-16
```

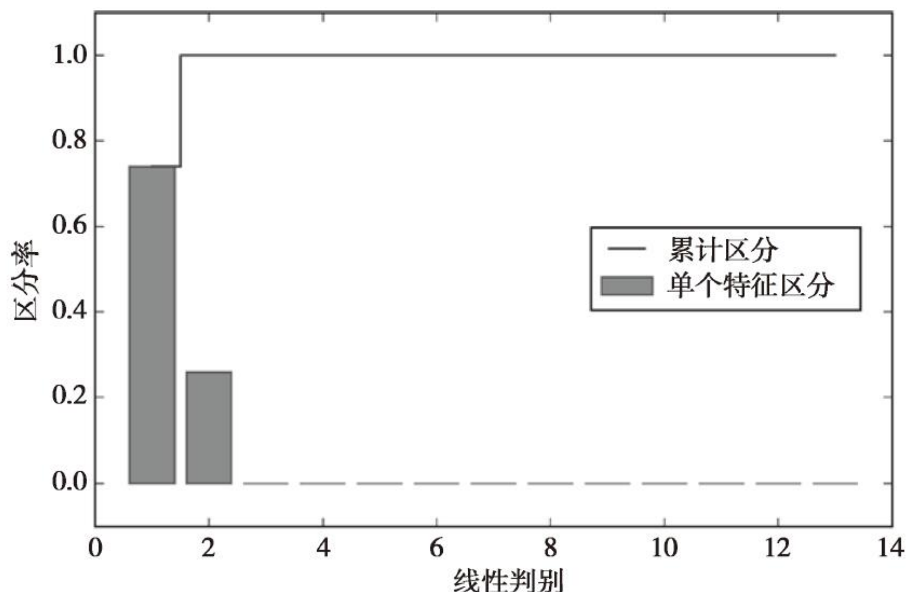
熟悉线性代数的读者应该知道：d×d维协方差矩阵的秩最大为d-1，而且确实可以发现，我们只得到了两个非零特征值（实际得到的第3～13个特征值并非完全为零，而是趋近于0的实数，这个结果是由NumPy浮点

运算导致的）。请注意，在极少的情况下可达到完美的共线性（所有样本的点落在一条直线上），这时协方差矩阵的秩为1，将导致矩阵只有一个含非零特征值的特征向量。

为了度量线性判别（特征向量）可以获取多少可区分类别的信息，与前面PCA小节中对累计方差的绘制类似，我们按照特征值降序绘制出特征对线性判别信息保持程度的图像。为了简便起见，我们在此使用了判定类别区分能力的相关信息discriminability。

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real, reverse=True)]
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, alpha=0.5, align='center',
...         label='individual "discriminability"')
>>> plt.step(range(1, 14), cum_discr, where='mid',
...          label='cumulative "discriminability"')
>>> plt.ylabel('"discriminability" ratio')
>>> plt.xlabel('Linear Discriminants')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.show()
```

从结果图像中可以看到，前两个线性判别几乎获取到了葡萄酒训练数据集中全部有用信息：



下面我们叠加这两个判别能力最强的特征向量列来构建转换矩阵W：

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w)
Matrix W:
 [[-0.0707 -0.3778]
 [ 0.0359 -0.2223]
 [-0.0263 -0.3813]
 [ 0.1875  0.2955]
 [-0.0033  0.0143]
 [ 0.2328  0.0151]
 [-0.7719  0.2149]
 [-0.0803  0.0726]
 [ 0.0896  0.1767]
 [ 0.1815 -0.2909]
 [-0.0631  0.2376]
 [-0.3794  0.0867]
 [-0.3355 -0.586 ]]
```

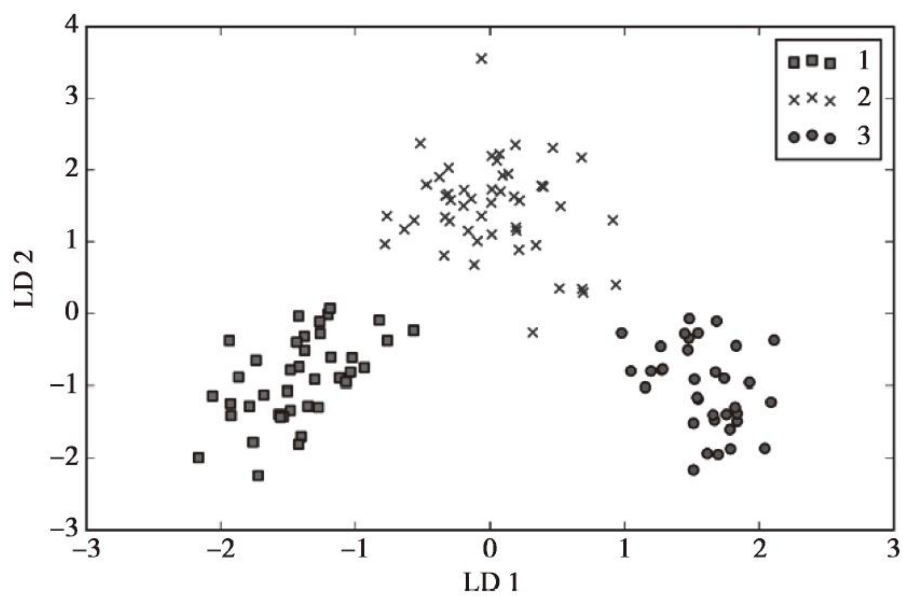
5.2.3 将样本映射到新的特征空间

通过上一小节中构建的转换矩阵W，我们可以通过乘积的方式对训练数据集进行转换：

$$X' = XW$$

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0],
...                 X_train_lda[y_train==l, 1],
...                 c=c, label=l, marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='upper right')
>>> plt.show()
```

通过结果图像可见，三个葡萄酒类在新的特征子空间上是线性可分的：



4
3
2
1
0
−1
−2
−3
LD 2
−3
−2
−1
0
1
2
3
LD 1
1
2
3

5.2.4　使用scikit-learn进行LDA分析

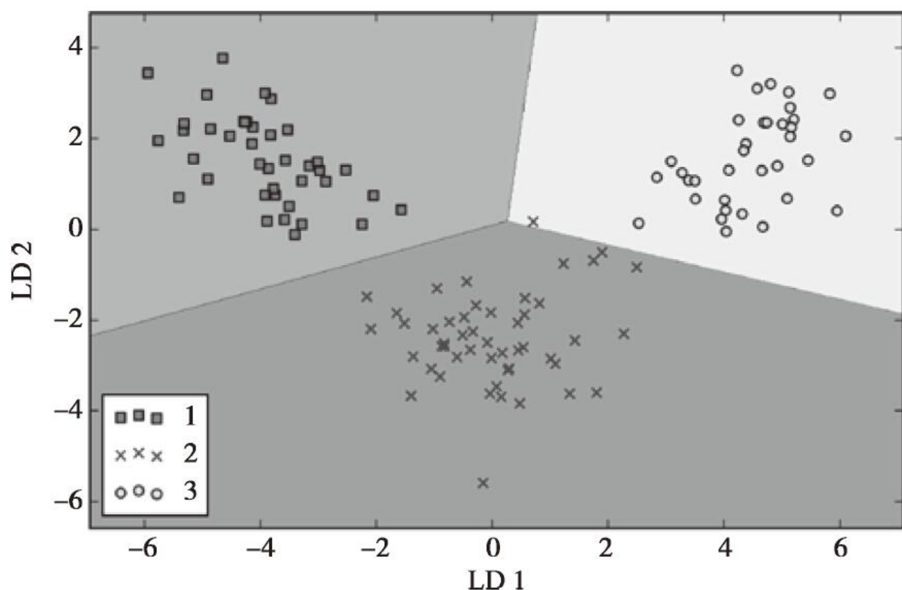
自已写代码逐步实现LDA，对于理解LDA内部的工作原理及其与PCA的差别是一种很好的体验。下面，我们来看看scikit-learn中对LDA类的实现：

```
>>> from sklearn.lda import LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

接下来，在将训练数据通过LDA进行转换后，我们来看一下逻辑斯谛回归在相对低维数据上的表现：

```
>>> lr = LogisticRegression()
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

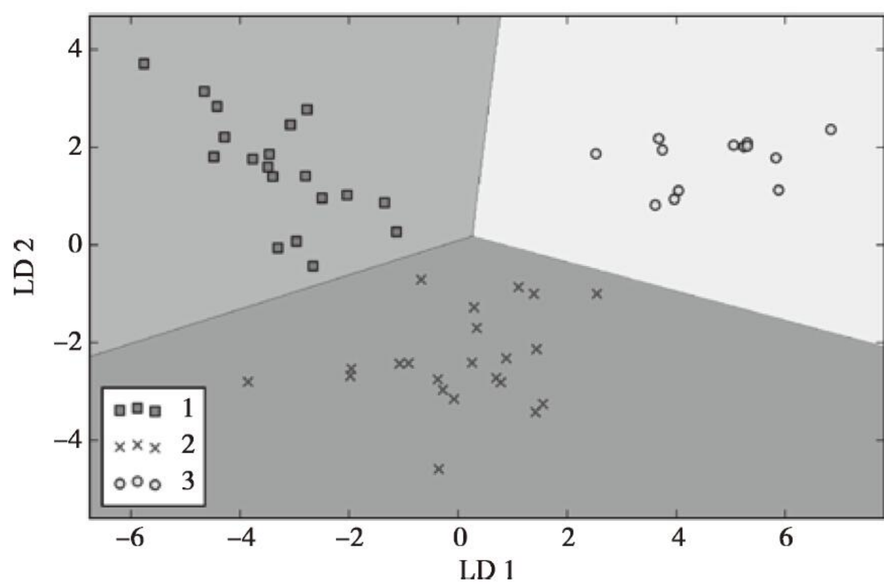
从结果图像中可以看到，逻辑斯谛回归模型只错误地判断了类别2中的一个样本：



通过降低正则化强度，我们或许可以对决策边界进行调整，以使得逻辑斯谛回归模型可以正确地对训练数据进行分类。下面，我们再来看一下模型在测试数据集上的效果：

```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

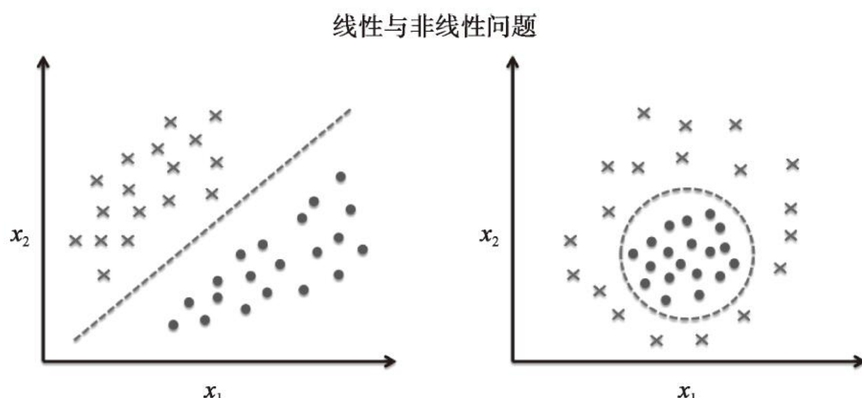
正如从结果图像中看到的那样，当我们使用只有两维的特征子空间来替代原始数据集中的13个葡萄酒特征时，逻辑斯谛回归在测试数据集上对样本的分类结果可谓完美：



LD 2
LD 1
1
2
3

5.3 使用核主成分分析进行非线性映射

许多机器学习算法都假定输入数据是线性可分的。感知器为了保证其收敛性，甚至要求训练数据是完美线性可分的。我们目前学习过的算法中，像Adaline、逻辑斯谛回归和（标准）支持向量机等，都将无法实现完美线性划分的原因归咎于噪声。然而，在现实世界中，大多数情况下我们面对的是非线性问题，针对此类问题，通过降维技术，如PCA和LDA等，将其转化为线性问题并不是最好的方法。在本小节中，我们将了解一下利用核技巧的PCA，或者称为核PCA，这与在第3章中我们介绍过的核支持向量机的概念有一定相关性。使用核PCA，我们将学习如何将非线性可分的数据转换到一个适合对其进行线性分类的新的低维子空间上。



5.3.1 核函数与核技巧

回忆一下我们在第3章中曾讨论过的基于核的支持向量机，通过将非线性可分问题映射到维度更高的特征空间，使其在新的特征空间上线性可分。为了将样本$x\in R^d$转换到维度更高的k维子空间，我们定义如下非线性映射函数φ：

$$\phi:\mathbb{R}^d\rightarrow\mathbb{R}^k\ (k>>d)$$

我们可以将φ看作是一个函数，它能够对原始特征进行非线性组合，以将原始的d维数据集映射到更高维的k维特征空间。例如：对于二维（d＝2）特征向量$x\in R^d$（x是包含d个特征的列向量）来说，可用如下映射将其转换到三维空间：

$$\boldsymbol{x}=[x_1,x_2]^{\mathrm{T}}$$
$$\downarrow\phi$$
$$\boldsymbol{z}=\left[x_1^2,\sqrt{2x_1x_2},x_2^2\right]^{\mathrm{T}}$$

换句话说，利用核PCA，我们可以通过非线性映射将数据转换到一个高维空间，然后在此高维空间中使用标准PCA将其映射到另外一个低维空间中，并通过线性分类器对样本进行划分（前提条件是，样本可根据输入空间的密度进行划分）。但是，这种方法的缺点是会带来高昂的计算成本，这也正是我们为什么要使用核技巧的原因。通过使用核技巧，我们可以在原始特征空间中计算两个高维特征空间中向量的相似度。

在更深入了解使用核技巧解决计算成本高昂的问题之前，我们先回顾

一下本章最初所介绍的标准PCA方法。两个特征k和j之间协方差的计算公式如下：

$$\sigma_{jk}=\frac{1}{n}\sum_{i=1}^{n}(x_j^{(i)}-\mu_j)(x_k^{(i)}-\mu_k)$$

由于在对特征做标准化处理后，其均值为0，例如：我们可将上述公式简化为：

$$\sigma_{jk}=\frac{1}{n}\sum_{i=1}^{n}x_j^{(i)}x_k^{(i)}$$

请注意，上述公式是两个特征值之间的协方差计算公式，下面给出计算协方差矩阵Σ的通用公式：

$$\Sigma=\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)\mathrm{T}}$$

Bernhard Schoellkopf提出了一种方法[1]，可以使用φ通过在原始特征空间上的非线性特征组合来替代样本间点积的计算：

$$\Sigma=\frac{1}{n}\sum_{i=1}^{n}\phi(\boldsymbol{x}^{(i)})\phi(\boldsymbol{x}^{(i)})^{\mathrm{T}}$$

为了求得此协方差矩阵的特征向量，也就是主成分，我们需要求解下述公式：

$$\Sigma\boldsymbol{v}=\lambda\boldsymbol{v}$$

$$\Rightarrow\frac{1}{n}\sum_{i=1}^{n}\phi(\boldsymbol{x}^{(i)})\phi(\boldsymbol{x}^{(i)})^{\mathrm{T}}\boldsymbol{v}=\lambda\boldsymbol{v}$$

$$\Rightarrow\boldsymbol{v}=\frac{1}{n\lambda}\sum_{i=1}^{n}\phi(\boldsymbol{x}^{(i)})\phi(\boldsymbol{x}^{(i)})^{\mathrm{T}}\boldsymbol{v}=\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{a}^{(i)}\phi(\boldsymbol{x}^{(i)})$$

其中，λ和v分别为协方差矩阵Σ的特征值和特征向量，这里的a可以通

过提取核（相似）矩阵K的特征向量来得到，具体内容在将后续段落中进行介绍。

核矩阵的推导过程如下：

首先，使用矩阵符号来表示协方差矩阵，其中φ（X）是一个n×k维的矩阵：

$$\Sigma=\frac{1}{n}\sum_{i=1}^{n}\phi(\boldsymbol{x}^{(i)})\ \phi\ (\boldsymbol{x}^{(i)})^{\mathrm{T}}=\frac{1}{n}\phi(\boldsymbol{X})^{\mathrm{T}}\phi(\boldsymbol{X})$$

现在，我们可以将特征向量的公式记为：

$$v=\frac{1}{\boldsymbol{n}}\sum_{i=1}^{n}\boldsymbol{a}^{(i)}\phi(\boldsymbol{x}^{(i)})=\lambda\phi(\boldsymbol{X})^{\mathrm{T}}\boldsymbol{a}$$

由于Σv = λv，我们可以得到：

$$\frac{1}{n}\phi(\boldsymbol{X})^{\mathrm{T}}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{\mathrm{T}}\boldsymbol{a}=\lambda\phi(\boldsymbol{X})^{\mathrm{T}}\boldsymbol{a}$$

两边同乘以φ（X），可得：

$$\frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{\mathrm{T}}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{\mathrm{T}}\boldsymbol{a}=\lambda\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{\mathrm{T}}\boldsymbol{a}$$
$$\Rightarrow\frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{\mathrm{T}}\boldsymbol{a}=\lambda\boldsymbol{a}$$
$$\Rightarrow\frac{1}{n}\boldsymbol{K}\boldsymbol{a}=\lambda\boldsymbol{a}$$

其中，K为相似（核）矩阵：

$$\boldsymbol{K}=\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{\mathrm{T}}$$

回顾一下3.4节的内容，通过核技巧，使用核函数K以避免使用φ来精确计算样本集合x中样本对之间的点积，这样我们就无需对特征向量进行

精确的计算：

$$\kappa(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)})=\phi(\boldsymbol{x}^{(i)})^{\mathrm{T}}\phi(\boldsymbol{x}^{(j)})$$

换句话说，通过核PCA，我们能够得到已经映射到各成分的样本，而不像标准PCA方法那样去构建一个转换矩阵。简单地说，可以将核函数（或者简称为核）理解为：通过两个向量点积来度量向量间相似度的函数。最常用的核函数有：

·多项式核：

$$\kappa(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)})=(\boldsymbol{x}^{(i)\mathrm{T}}\boldsymbol{x}^{(j)}+\theta)^{p}$$

其中，阈值θ和幂的值p需自行定义。

·双曲正切（sigmoid）核：

$$\kappa(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)})=\mathrm{thah}(\eta\boldsymbol{x}^{(i)\mathrm{T}}\boldsymbol{x}^{(j)}+\theta)$$

·径向基核函数（Radial Basis Function，RBF）或者称为高斯核函数，我们将在下一小节的示例中用到：

$$\kappa(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)})=\exp\left(-\frac{\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\|^{2}}{2\sigma^{2}}\right)$$

也可以写作：

$$\kappa(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)})=\exp(-\gamma\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\|^{2})$$

综合上述讨论，我们可以通过如下三个步骤来实现一个基于RBF核的PCA：

1）为了计算核（相似）矩阵k，我们需要做如下计算：

$$\kappa(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)})=\exp(-\gamma\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\|^2)$$

我们需要计算任意两样本对之间的值：

$$\boldsymbol{K}=\begin{bmatrix}\kappa(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(1)}) & \kappa(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(2)}) & \cdots & \kappa(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(n)})\\ \kappa(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(1)}) & \kappa(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(2)}) & \cdots & \kappa(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(n)})\\ \vdots & \vdots & \ddots & \vdots\\ \kappa(\boldsymbol{x}^{(n)},\boldsymbol{x}^{(1)}) & \kappa(\boldsymbol{x}^{(d)},\boldsymbol{x}^{(2)}) & \cdots & \kappa(\boldsymbol{x}^{(n)},\boldsymbol{x}^{(n)})\end{bmatrix}$$

例如：如果数据集包含100个训练样本，将得到一个100×100维的对称核矩阵。

2）通过如下公式进行计算，使核矩阵k更为聚集：

$$\boldsymbol{K}'=\boldsymbol{K}-1_n\boldsymbol{K}-\boldsymbol{K}1_n+1_n\boldsymbol{K}1_n$$

其中，l_n 是一个 $n\times n$ 维的矩阵（与核矩阵维度相同），其所有的值均为$\frac{1}{n}$。

3）将聚集后的核矩阵的特征值按照降序排列，选择前k个特征值所对应的特征向量。与标准PCA不同，这里的特征向量不是主成分轴，而是将样本映射到这些轴上。

至此，读者可能会感到困惑：为什么要在第2步中对核矩阵进行聚集处理？我们曾经假定，数据需要经过标准化处理，当在生成协方差矩阵并通过φ以非线性特征的组合替代点积时，所有特征的均值为0。由此，在第2步中聚集核矩阵就显得很有必要，因为我们并没有精确计算新的特征空间，而且也不能确定新特征空间的中心在零点。

下一小节中，我们将基于这三个步骤使用Python实现核PCA。

[1] B. Scholkopf, A. Smola, and K.-R. Muller. Kernel Principal Component Analysis. pages 583–588, 1997.

5.3.2 使用Python实现核主成分分析

在前面的小节中，我们讨论了核PCA相关的核心概念。现在根据之前总结过的实现核PCA方法的三个步骤，使用Python实现基于RBF核的PCA。借助于SciPy和NumPy的函数，我们将会看到，实现核PCA实际上很简单。

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_samples, n_features]

    gamma: float
      Tuning parameter of the RBF kernel

    n_components: int
      Number of principal components to return

    Returns
    ------------
     X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
       Projected dataset

    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
```

```
    K = exp(-gamma * mat_sq_dists)

    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtaining eigenpairs from the centered kernel matrix
    # numpy.eigh returns them in sorted order
    eigvals, eigvecs = eigh(K)

    # Collect the top k eigenvectors (projected samples)
    X_pc = np.column_stack((eigvecs[:, -i]
                            for i in range(1, n_components + 1)))

    return X_pc
```

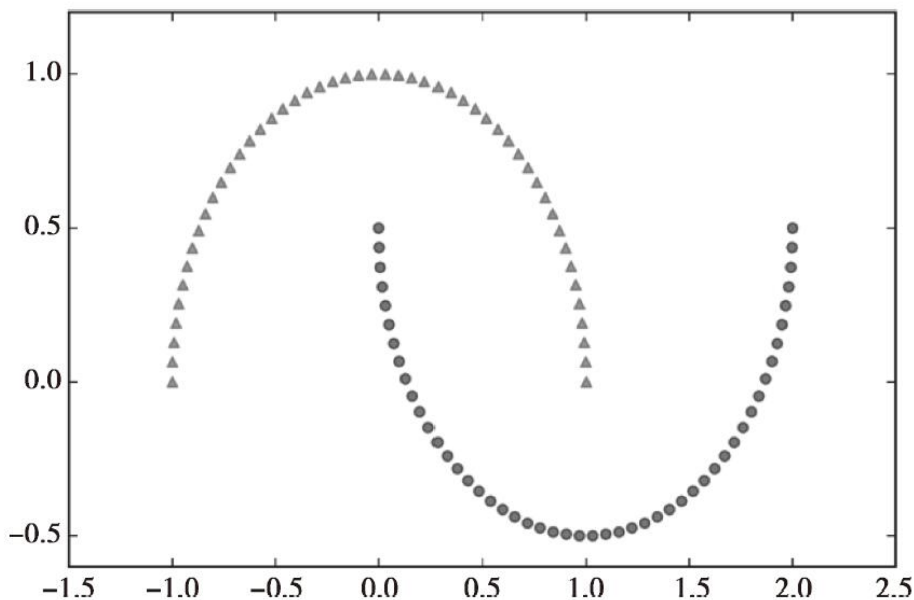
采用RBF核函数实现的PCA进行降维时存在一个问题，就是我们必须指定先验参数r需要通过实验来找到一个合适的r值，最好是通过参数调优算法来确定，例如网格搜索法，我们将在第6章中对其进行深入探讨。

1.示例一：分离半月形数据

现在，我们将实现的rbf_kernel_pca方法应用于非线性示例数据集。我们首先创建一个包含100个样本点的二维数据集，以两个半月形状表示：

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

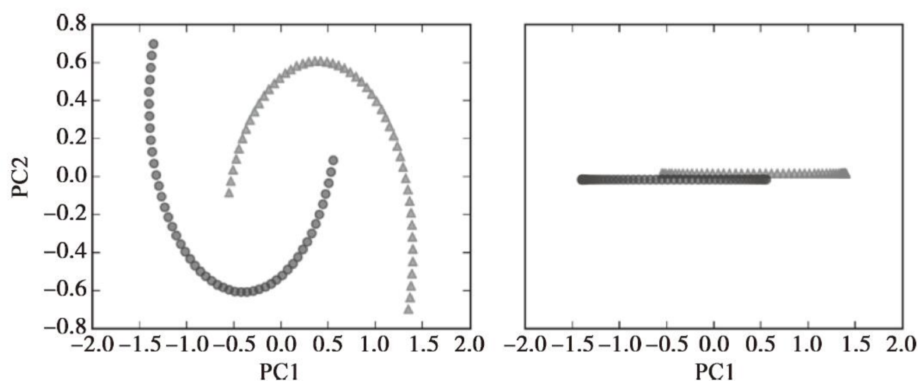
出于演示的需要，使用三角符号标识的表示一个类别中的样本，使用圆形符号标识的表示另一类别的样本：



显然，这两个半月形不是线性可分的，而我们的目标是通过核PCA将这两个半月形数据展开，使得数据集成为适用于某一线性分类器的输入数据。首先，我们通过标准的PCA将数据映射到主成分上，并观察其形状：

```
>>> from sklearn.decomposition import PCA
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

很明显，经过标准PCA的转换后，线性分类器未必能很好地发挥其作用：



请注意，当我们仅绘制第一主成分的图像时（见右子图），我们分别将三角形和圆形代表的样本向上或向下做了轻微调整，以更好地展示类间重叠。

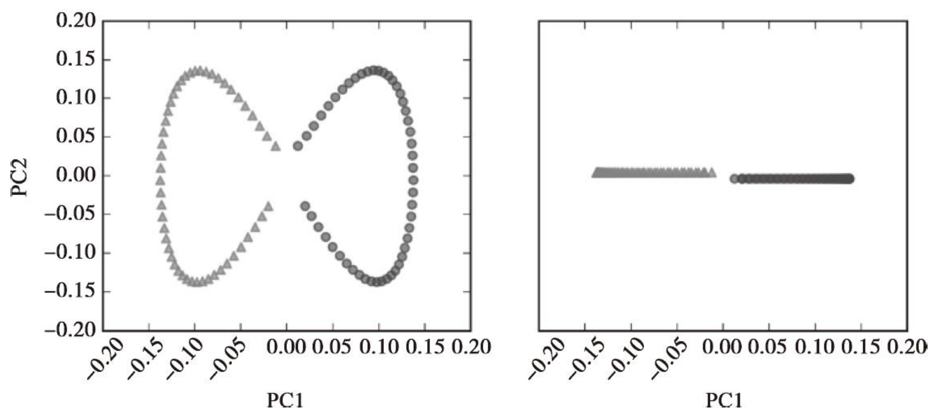
请注意，PCA是无监督方法，与LDA相比，它在使得方差最大化的过程中未使用类标信息。出于增强可视化效果的考虑，为了显示分类的程度，我们才在此使用了三角形和圆形符号。

现在我们将使用前一小节中实现的核PCA函数：rbf_kernel_pca：

```
>>> from matplotlib.ticker import FormatStrFormatter
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
```

```
...                   color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...                   color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...                   color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> ax[0].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> ax[1].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
>>> plt.show()
```

可以看到，两个类别（圆形和三角形）此时是线性可分的，这使得转换后的数据适合作为线性分类器的训练数据集：



不过，对于可调整参数γ，没有一个通用的值使其适用于不同的数据集。针对给定问题找到一个适宜的参数值需要通过实验来解决。在第6章中，我们将讨论可自动进行参数优化等任务的技术。这里，我使用了一个已有的能够生成良好结果的值。

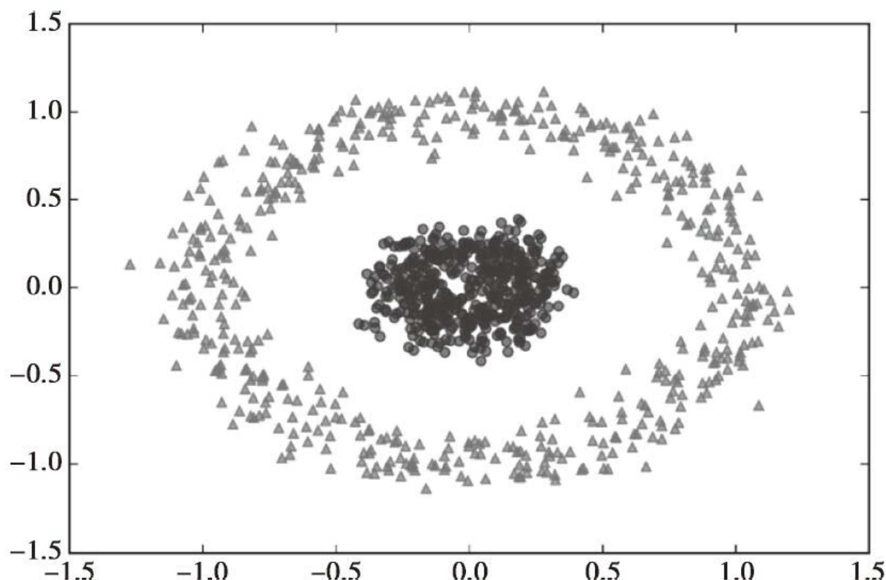
2.示例二：分离同心圆

在上一小节，我们演示了如何通过核PCA分离半月形数据。既然我们已经投入了如此多的精力去理解核PCA的概念，就再看一下另外一个关于非线性问题的有趣例子——同心圆。

代码如下：

```
>>> from sklearn.datasets import make_circles
>>> X, y = make_circles(n_samples=1000,
...             random_state=123, noise=0.1, factor=0.2)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.show()
```

同样，我们假设了一个涉及两个类别的问题，三角形和圆形分别标识不同类别的样本：

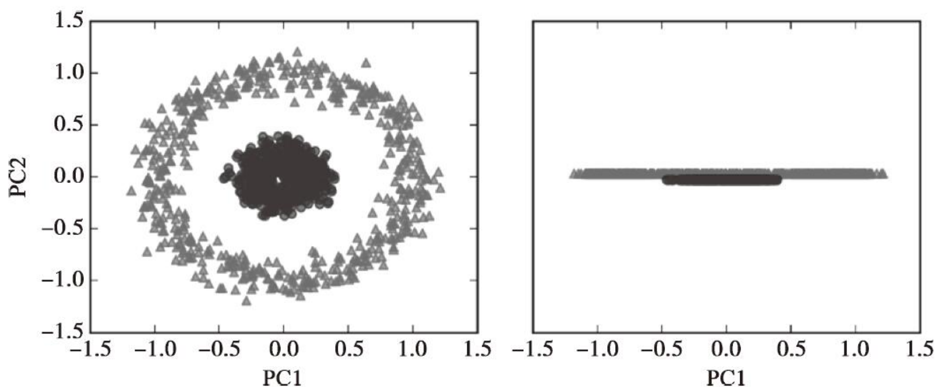


首先使用标准PCA方法，以便将其结果与基于RBF核的PCA生成的结

果进行比较：

```
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...                 color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...                 color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
...                color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
...                 color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

再一次发现，通过标准PCA无法得到适合于线性分类器的训练数据：

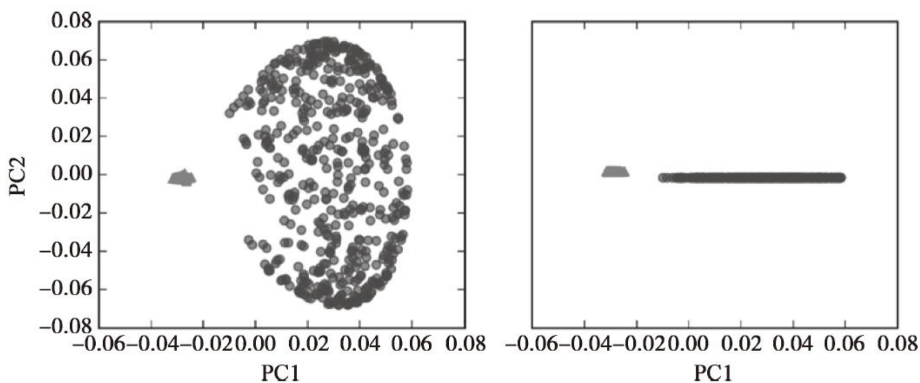


给定一个合适的γ值，来看看基于RBF的核PCA实现能否得到令人满意的结果：

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
```

```
...                   color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...                   color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
...                   color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
...                   color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.show()
```

基于RBF的核PCA再一次将数据映射到了一个新的子空间中，使两个类别变得线性可分：



5.3.3 映射新的数据点

在上述两个核PCA应用的例子（半月形和同心圆）中，我们都将单一数据集映射到一个新的特征上。但在实际应用中，我们可能需要转换多个数据集，例如，训练数据、测试数据，以及在完成模型构建和评估后所要收集的新样本。本节将介绍如何映射训练数据集以外的数据点。

在本章开始时介绍过的标准PCA方法中，我们通过转换矩阵和输入样本之间的点积来对数据进行映射；映射矩阵的列是协方差矩阵中k个最大特征值所对应的特征向量（v）。现在的问题是：如何将此概念应用于核PCA？回忆一下核PCA的原理可以记得，我们从聚集核矩阵（不是协方差矩阵）中得到了特征向量（a），这意味着样本已经映射到了主成分轴v。由此，如果我们希望将新的样本（x′）映射到此主成分轴，需要进行如下计算：

$$\phi(\boldsymbol{x}')^{\mathrm{T}}\boldsymbol{v}$$

幸运的是，我们可以使用核技巧，这样就无需精确计算映射φ（x′）^{T}v。然而值得注意的是：与标准PCA相比，核PCA是一种基于内存的方法，这意味着每次映射新的样本前，必须再次使用原始训练数据。我们需要计算训练数据集中每一个训练样本和新样本x′之间的RBF核（相似度）：

$$\begin{aligned}\phi(\boldsymbol{x}')^{\mathrm{T}}\boldsymbol{v} &= \sum_i a^{(i)}\phi(\boldsymbol{x}')^{\mathrm{T}}\phi(\boldsymbol{x}^{(i)}) \\ &= \sum_i a^{(i)}k(\boldsymbol{x}',\boldsymbol{x}^{(i)})^{\mathrm{T}}\end{aligned}$$

其中，核矩阵K的特征向量a及特征值λ需满足如下等式：

$$Ka = \lambda a$$

在完成新样本与训练数据集内样本间相似度的计算后，我们还需通过特征向量对应的特征值来对其进行归一化处理。可以通过修改早前实现过的rbf_kernel_pca函数来让其返回核矩阵的特征值：

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_samples, n_features]

    gamma: float
      Tuning parameter of the RBF kernel

    n_components: int
      Number of principal components to return

    Returns
    ------------
     X_pc: {NumPy ndarray}, shape = [n_samples, k_features]
       Projected dataset

     lambdas: list
       Eigenvalues

    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)

    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtaining eigenpairs from the centered kernel matrix
    # numpy.eigh returns them in sorted order
    eigvals, eigvecs = eigh(K)

    # Collect the top k eigenvectors (projected samples)
```

```
    alphas = np.column_stack((eigvecs[:,-i]
                   for i in range(1,n_components+1)))

    # Collect the corresponding eigenvalues
    lambdas = [eigvals[-i] for i in range(1,n_components+1)]

    return alphas, lambdas
```

至此，我们可以创建一个新的半月形数据集，并使用更新过的RBF核PCA实现来将其映射到一个一维的子空间上：

```
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> alphas, lambdas =rbf_kernel_pca(X, gamma=15, n_components=1)
```

为了确保我们已经完成了实现新样本映射的代码，假定半月形数据集中的第26个点是一个新的数据点x′，现在要将其映射到新的子空间中：

```
>>> x_new = X[25]
>>> x_new
array([ 1.8713187 ,  0.00928245])
>>> x_proj = alphas[25] # original projection
>>> x_proj
array([ 0.07877284])
>>> def project_x(x_new, X, gamma, alphas, lambdas):
...     pair_dist = np.array([np.sum(
...                  (x_new-row)**2) for row in X])
...     k = np.exp(-gamma * pair_dist)
... return k.dot(alphas / lambdas)
```

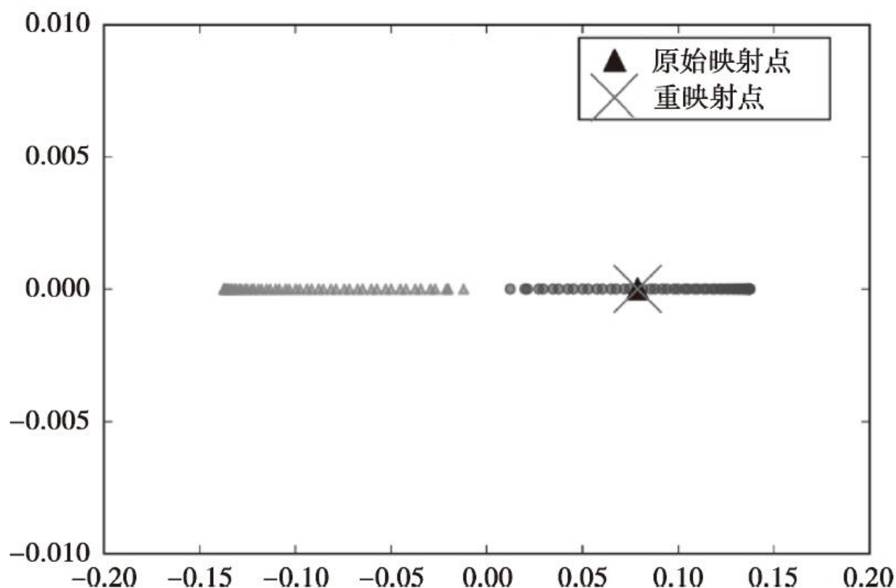
通过执行下面的代码，我们可以重现原始映射。使用project_x函数，还可以映射新的数据样本。代码如下：

```
>>> x_reproj = project_x(x_new, X,
...       gamma=15, alphas=alphas, lambdas=lambdas)
>>> x_reproj
array([ 0.07877284])
```

最后，将第一主成分上的映射进行可视化：

```
>>> plt.scatter(alphas[y==0, 0], np.zeros((50)),
...             color='red', marker='^',alpha=0.5)
>>> plt.scatter(alphas[y==1, 0], np.zeros((50)),
...             color='blue', marker='o', alpha=0.5)
>>> plt.scatter(x_proj, 0, color='black',
...             label='original projection of point X[25]',
...             marker='^', s=100)
>>> plt.scatter(x_reproj, 0, color='green',
...             label='remapped point X[25]',
...             marker='x', s=500)
>>> plt.legend(scatterpoints=1)
>>> plt.show()
```

从下图可见，我们将样本x′正确映射到了第一主成分上：



5.3.4 scikit-learn中的核主成分分析

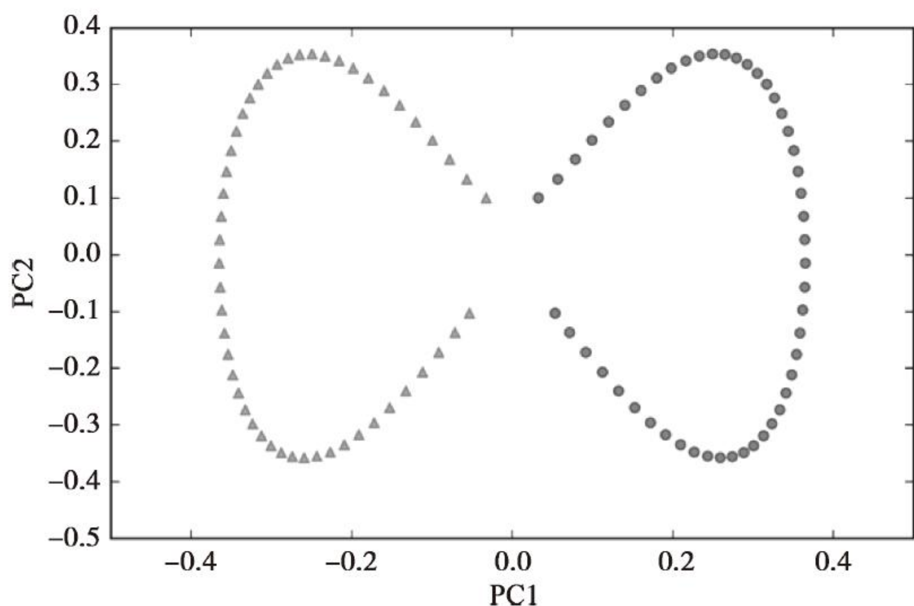
scikit-learn的sklearn.decomposition子模块中已经实现了一种核PCA类。其使用方法与标准PCA类类似，我们可以通过kernel参数来选择不同的核函数：

```
>>> from sklearn.decomposition import KernelPCA
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> scikit_kpca = KernelPCA(n_components=2,
...                         kernel='rbf', gamma=15)
>>> X_skernpca = scikit_kpca.fit_transform(X)
```

为了验证得到的结果与我们自己实现的核PCA是否一致，我们来绘制将半月形数据映射到前两个主成分的图像：

```
>>> plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.show()
```

从结果图像中可见，通过scikit-learn中KernelPCA得到的结果与我们自己实现得到的结果一致：



scikit-learn实现了一些高级的非线性降维技术，这些内容已经超出了本书的范围。读者可以通过链接http://scikit-learn.org/stable/modules/manifold.html来了解相关内容概述及其示例。

本章小结

在本章中，读者学习了三种不同的基于特征提取的基本降维技术：标准PCA、LDA，以及核PCA。使用PCA，我们可以在忽略类标的情况下，将数据映射到一个低维的子空间上，并沿正交的特征坐标方向使方差最大化。与PCA不同，LDA是一种监督降维技术，这意味着：在线性特征空间中尝试使得类别最大可分时，需要使用训练数据集中的类别信息。最后，我们学习了核PCA，通过核PCA可以将非线性数据集映射到一个低维的特征空间中，使得数据线性可分。

在掌握了这些数据预处理的基本技术后，读者可以准备进入下一章，学习如何有效地组合不同的预处理技术，以及模型性能评估等方面的内容。

第6章　模型评估与参数调优实战

在前面的章节中，读者学习了用于分类的机器学习基础算法，以及在使用算法前的数据预处理方法。在本章中，我们将使用代码进行实践，通过对算法进行调优来构建性能良好的机器学习模型，并对模型的性能进行评估。我们将学习如下内容：

·模型性能的无偏估计

·处理机器学习算法中常见问题

·机器学习模型调优

·使用不同的性能指标评估预测模型

6.1　基于流水线的工作流

我们在前面章节中学习了不同的数据预处理技术，无论是第4章中介绍的用于特征缩放的标准化方法，还是第5章中介绍的用于数据压缩的主成分分析等，我们在使用训练数据对模型进行拟合时就得到了一些参数，但将模型用于新数据时需重设这些参数。本节读者将学到一个方便使用的工具：scikit-learn中的Pipline类。它使得我们可以拟合出包含任意多个处理步骤的模型，并将模型用于新数据的预测。

6.1.1 加载威斯康星乳腺癌数据集

本章我们将使用威斯康星乳腺癌（Breast Cancer Wisconsin）数据集进行讲解，此数据集共包含569个恶性或者良性肿瘤细胞样本。数据集的前两列分别存储了样本唯一的ID以及对样本的诊断结果（M代表恶性，B代表良性）。数据集的3～32列包含了30个从细胞核照片中提取、用实数值标识的特征，它们可以用于构建判定模型，对肿瘤是良性还是恶性做出预测。威斯康星乳腺癌数据集已经存储在UCI机器学习数据集库中，关于此数据集更多的信息请访问链接：https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+（Diagnostic）。

在本节，我们通过三个步骤来读取数据集，并将其划分为训练数据集和测试数据集。

1）使用pandas从UCI网站直接读取数据集：

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/breast-cancer-wisconsin/wdbc.data',
header=None)
```

2）接下来，将数据集的30个特征的赋值给一个NumPy的数组对象X。使用scikit-learn中的LabelEncoder类，我们可以将类标从原始的字符串表示（M或者B）转换为整数：

```
>>> from sklearn.preprocessing import LabelEncoder
>>> X = df.loc[:, 2:].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

转换后的类标（诊断结果）存储在一个数组y中，此时恶性肿瘤和良性肿瘤分别被标识为类1和类0，我们可以通过调用LabelEncoder的transform方法来显示虚拟类标（0和1）：

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

3）在构建第一个流水线模型前，先将数据集划分为训练数据集（原始数据集80%的数据）和一个单独的测试数据集（原始数据集20%的数据）：

```
>>> from sklearn.cross_validation import train_test_split
>>> X_train, X_test, y_train, y_test = \
...      train_test_split(X, y, test_size=0.20, random_state=1)
```

6.1.2 在流水线中集成数据转换及评估操作

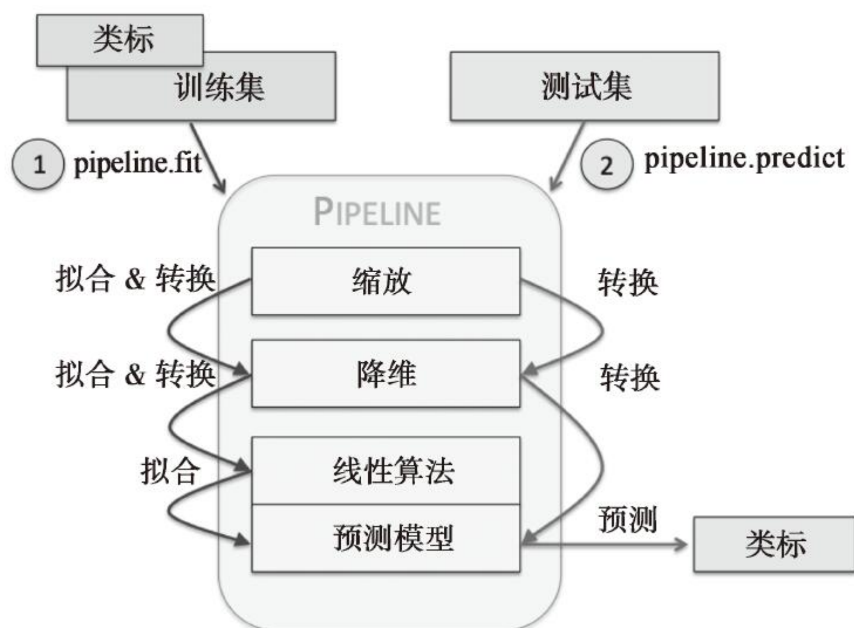
通过前面章节的学习，我们了解到：出于性能优化的目的，许多学习算法要求将不同特征的值缩放到相同的范围。这样，我们在使用逻辑斯谛回归模型等线性分类器分析威斯康星乳腺癌数据集之前，需要对其特征列做标准化处理。此外，我们还想通过第5章中介绍过的主成分分析（PCA）——使用特征抽取进行降维的技术，将最初的30维数据压缩到一个二维的子空间上。我们无需在训练数据集和测试数据集上分别进行模型拟合、数据转换，而是通过流水线将StandardScaler、PCA，以及LogisticRegression对象串联起来：

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import Pipeline
>>> pipe_lr = Pipeline([('scl', StandardScaler()),
...             ('pca', PCA(n_components=2)),
...             ('clf', LogisticRegression(random_state=1))])
>>> pipe_lr.fit(X_train, y_train)
>>> print('Test Accuracy: %.3f' % pipe_lr.score(X_test, y_test))
Test Accuracy: 0.947
```

Pipeline对象采用元组的序列作为输入，其中每个元组中的第一个值为一个字符串，它可以是任意的标识符，我们通过它来访问流水线中的元素，而元组的第二个值则为scikit-learn中的一个转换器或者评估器。

流水线中包含了scikit-learn中用于数据预处理的类，最后还包括一个评估器。在前面的示例代码中，流水线中有两个预处理环节，分别是用于数据缩放和转换的StandardScaler及PCA，最后还有一个作为评估器的逻辑斯谛回归分类器。当在流水线pipe_lr上执行fit方法时，StandardScaler

会在训练数据上执行fit和transform操作，经过转换后的训练数据将传递给流水线上的下一个对象——PCA。与前面的步骤类似，PCA会在前一步转换后的输入数据上执行fit和transform操作，并将处理过的数据传递给流水线中的最后一个对象——评估器。我们应该注意到：流水线中没有限定中间步骤的数量。流水线的工作方式可用下图来描述：



6.2　使用k折交叉验证评估模型性能

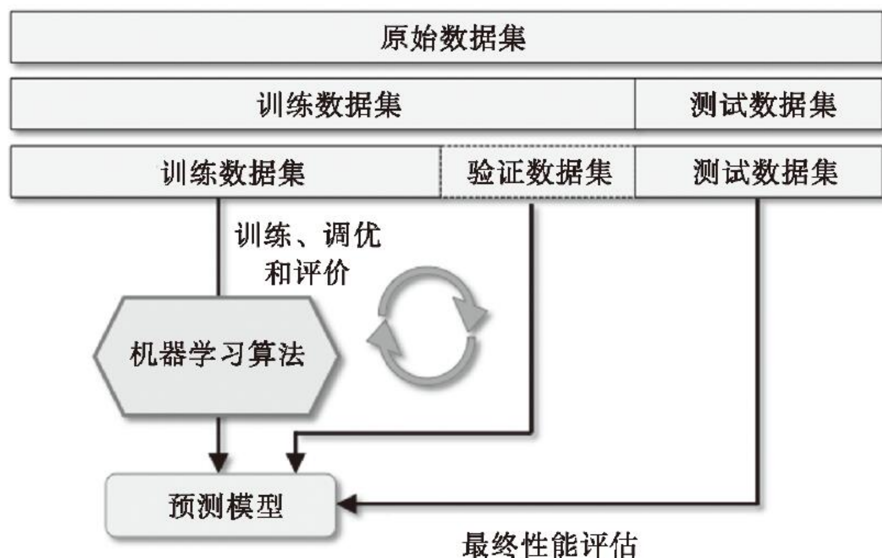
构建机器学习模型的一个关键步骤就是在新数据上对模型的性能进行评估。现在假设我们使用训练数据集对模型进行拟合，并且使用同样的数据对其性能进行评估。回忆一下3.3.4节，如果一个模型过于简单，将会面临欠拟合（高偏差）的问题，而模型基于训练数据构造得过于复杂，则会导致过拟合（高方差）问题。为了在偏差和方差之间找到可接受的折中方案，我们需要对模型进行评估。在本节，读者将学到有用的交叉验证技术：holdout交叉验证（holdout cross-validation）和k折交叉验证（k-fold cross-validation），借助于这两种方法，我可以得到模型泛化误差的可靠估计，即模型在新数据上的性能表现。

6.2.1 holdout方法

holdout交叉验证是评估机器学习模型泛化性能的一个经典且常用的方法。通过holdout方法，我们将最初的数据集划分为训练数据集和测试数据集：前者用于模型的训练，而后者则用于性能的评估。然而，在典型的机器学习应用中，为进一步提高模型在预测未知数据上的性能，我们还要对不同参数设置进行调优和比较。该过程称为模型选择（model selection），指的是针对给定分类问题我们调整参数以寻求最优值（也称为超参，hyperparameter）的过程。

但是，如果我们在模型选择过程中不断重复使用相同的测试数据，它们可看作训练数据的一部分，模型更易于过拟合。尽管存在此问题，许多人仍旧在模型选择过程中使用这些测试数据，这对机器学习来说不是一种好的做法。

使用holdout进行模型选择更好的方法是将数据划分为三个部分：训练数据集、验证数据集和测试数据集。训练数据集用于不同模型的拟合，模型在验证数据集上的性能表现作为模型选择的标准。使用模型训练及模型选择阶段不曾使用的数据作为测试数据集的优势在于：评估模型应用于新数据上能够获得较小偏差。下图介绍了holdout交叉验证的一些概念。交叉验证中，在使用不同参数值对模型进行训练之后，我们使用验证数据集反复进行模型性能的评估。一旦参数优化获得较为满意的结果，我们就可以在测试数据集上对模型的泛化误差进行评估：



holdout方法的一个缺点在于：模型性能的评估对训练数据集划分为训练及验证子集的方法是敏感的；评价的结果会随样本的不同而发生变化。在下一小节中，我们将学习一种鲁棒性更好的性能评价技术：k折交叉验证，我们将在k个训练数据子集上重复holdout方法k次。

6.2.2 k折交叉验证

在k折交叉验证中，我们不重复地随机将训练数据集划分为k个，其中k-1个用于模型的训练，剩余的1个用于测试。重复此过程k次，我们就得到了k个模型及对模型性能的评价。

读者可能对有放回抽样和无放回抽样不太熟悉，我们在此简单做个介绍。假设我们在玩彩票游戏，游戏的规则是从一个盒子中随机抽取数字。盒子中包含5个数字0、1、2、3和4，每轮我们都只抽取出其中的一个。在第一轮中，我们抽到某个特定数字的概率为1/5。现在我们采用无放回抽样，也就是完成数字的抽取后，不再将其放回盒子。这时，抽到某个特定数字的概率依赖于上一轮的抽取结果。例如，如果盒子中剩余的数字为0、1、2和4，则下一轮抽取中得到数字0的概率应为1/4。

不过，在有放回随机抽样中，我们会将抽取到的数字再放回到盒子中，这样每一轮中抽取到某个特定数字的概率不发生改变，同时我们也可以多次抽取到同一个数字。换句话说，在有放回抽样中，样本（数字）是相互独立的，它们的协方差为0。例如，5次随机抽样的结果可能如下：

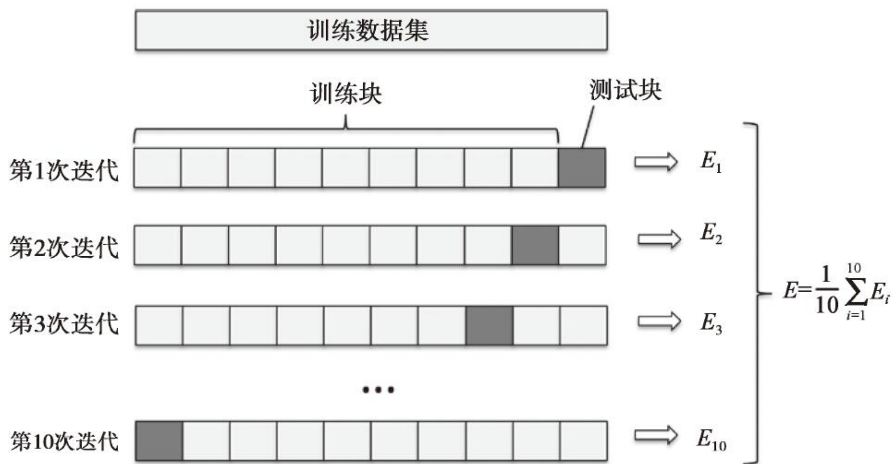
·无放回随机抽样：2，1，3，4，0

·有放回随机抽样：1，3，3，4，1

基于这些独立且不同的数据子集上得到的模型性能评价结果，我们可以计算出其平均性能，与holdout方法相比，这样得到的结果对数据划分方法的敏感性相对较低。通常情况下，我们将k折交叉验证用于模型的调

优，也就是找到使得模型泛化性能最优的超参值。一旦找到了满意的超参值，我们就可以在全部的训练数据上重新训练模型，并使用独立的测试数据集对模型性能做出最终评价。

由于k折交叉验证使用了无重复抽样技术，该方法的优势在于（每次迭代过程中）每个样本点只有一次被划入训练数据集或测试数据集的机会，与holdout方法相比，这将使得模型性能的评估具有较小的方差。下图总结了k折交叉验证的相关概念，其中k＝10。训练数据集被划分为10块，在10次迭代中，每次迭代都将9块用于训练，剩余的1块作为测试集用于模型的评估。此外，每次迭代中得到的性能评价指标E_i（例如，分类的准确性或者误差）可用来计算模型的估计平均性能E：



k折交叉验证中k的标准值为10，这对大多数应用来说都是合理的。但是，如果训练数据集相对较小，那就有必要加大k的值。如果我们增大k的值，在每次迭代中将会有更多的数据用于模型的训练，这样通过计算各性能评估结果的平均值对模型的泛化性能进行评价时，可以得到较小的偏

差。但是，k值的增加将导致交叉验证算法运行时间延长，而且由于各训练块间高度相似，也会导致评价结果方差较高。另一方面，如果数据集较大，则可以选择较小的k值，如k = 5，这不光能够降低模型在不同数据块上进行重复拟合和性能评估的计算成本，还可以在平均性能的基础上获得对模型的准确评估。

k折交叉验证的一个特例就是留一（leave-ont-out，LOO）交叉验证法。在LOO中，我们将数据子集划分的数量等同于样本数（k = n），这样每次只有一个样本用于测试。当数据集非常小时，建议使用此方法进行验证。

分层k折交叉验证对标准k折交叉验证做了稍许改进，它可以获得偏差和方差都较低的评估结果，特别是类别比例相差较大时，详见R.Kohavi等人的研究[1]。在分层交叉验证中，类别比例在每个分块中得以保持，这使得每个分块中的类别比例与训练数据集的整体比例一致，下面通过scikit-learn中的StratifiedKFold迭代器来演示：

```
>>> import numpy as np
>>> from sklearn.cross_validation import StratifiedKFold
>>> kfold = StratifiedKFold(y=y_train,
...                         n_folds=10,
...                         random_state=1)
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print('Fold: %s, Class dist.: %s, Acc: %.3f' % (k+1,
```

```
...                    np.bincount(y_train[train]), score))
Fold: 1, Class dist.: [256 153], Acc: 0.891
Fold: 2, Class dist.: [256 153], Acc: 0.978
Fold: 3, Class dist.: [256 153], Acc: 0.978
Fold: 4, Class dist.: [256 153], Acc: 0.913
Fold: 5, Class dist.: [256 153], Acc: 0.935
Fold: 6, Class dist.: [257 153], Acc: 0.978
Fold: 7, Class dist.: [257 153], Acc: 0.933
Fold: 8, Class dist.: [257 153], Acc: 0.956
Fold: 9, Class dist.: [257 153], Acc: 0.978
Fold: 10, Class dist.: [257 153], Acc: 0.956
>>> print('CV accuracy: %.3f +/- %.3f' % (
...                    np.mean(scores), np.std(scores)))
CV accuracy: 0.950 +/- 0.029
```

首先，我们用训练集中的类标y_train来初始化sklearn.cross_validation模块下的Stratified-Kfold迭代器，并通过n_folds参数来设置块的数量。当我们使用kfold迭代器在k个块中进行循环时，使用train中返回的索引去拟合本章开始时所构建的逻辑斯谛回归流水线。通过pile_lr流水线，我们可以保证每次迭代中样本都得到适当的缩放（如标准化）。然后使用test索引计算模型的准确率，将其存储在score列表中，用于计算平均准确率以及性能评估标准差。

尽管之前的代码清楚地介绍了k折交叉验证的工作方式，scikit-learn中同样也实现了k折交叉验证评分的计算，这使得我们可以更加高效地使用分层k折交叉验证对模型进行评估：

```
>>> from sklearn.cross_validation import cross_val_score
>>> scores = cross_val_score(estimator=pipe_lr,
...                          X=X_train,
...                          y=y_train,
...                          cv=10,
...                          n_jobs=1)
>>> print('CV accuracy scores: %s' % scores)
CV accuracy scores: [ 0.89130435  0.97826087  0.97826087
                      0.91304348  0.93478261  0.97777778
                      0.93333333  0.95555556  0.97777778
                         0.95555556]
>>> print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
np.std(scores)))
CV accuracy: 0.950 +/- 0.029
```

cross_val_score方法具备一个极为有用的特点，它可以将不同分块的性能评估分布到多个CPU上进行处理。如果将n_jobs参数的值设为1，则与例子中使用的StratifiedKFold类似，只使用一个CPU对性能进行评估。如果设定n_jobs = 2，我们可以将10轮的交叉验证分布到两块CPU（如果使用的计算机支持）上进行，如果设置n_jobs = -1，则可利用计算机所有的CPU并行地进行计算。

关于交叉验证中泛化性能方差的估计已经超出了本书的讨论范围，不过读者可以参阅M.Markaou等人的论文[2]，该文章对上述问题做了精彩的描述。

同时，读者也可以了解其他的交叉验证技术，如.632 Bootstrap交叉验证（.632 Bootstrap cross-validation）方法[3]。

[1] R.Konavi et al.A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection.In Ijcai,volume 14，pages 1137-1145，1995.

[2] M. Markatou, H. Tian, S. Biswas, and G. M. Hripcsak. Analysis of

Variance of Cross-validation Estimators of the Generalization Error. Journal of Machine Learning Research, 6:1127–1168, 2005.

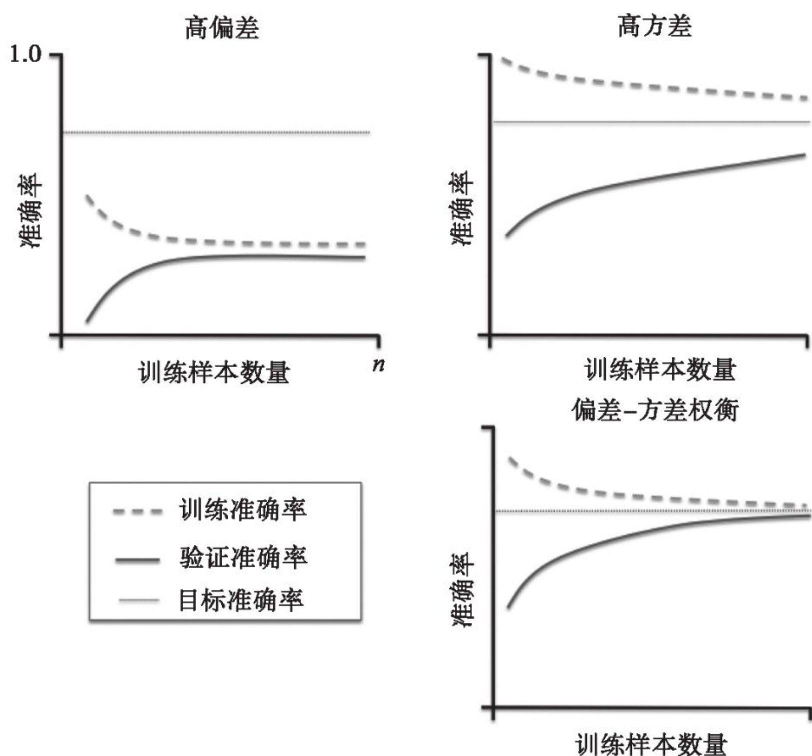
[3] B. Efron and R. Tibshirani. Improvements on Cross-validation ：The 632+ Bootstrap Method. Journal of the American Statistical Association，92(438):548—560，1997.

6.3 通过学习及验证曲线来调试算法

在本节，我们将学习两个有助于提高学习算法性能的简单但功能强大的判定工具：学习曲线（learning curve）与验证曲线（validation curve）。在下一小节中，我们将讨论如何使用学习曲线来判定学习算法是否面临过拟合（高方差）或欠拟合（高偏差）问题。此外，我们还将了解一下验证曲线，它可以帮助我们找到学习算法中的常见问题。

6.3.1　使用学习曲线判定偏差和方差问题

如果一个模型在给定训练数据上构造得过于复杂——模型中有太多的自由度或者参数——这时模型可能对训练数据过拟合，而对未知数据泛化能力低下。通常情况下，收集更多的训练样本有助于降低模型的过拟合程度。但是，在实践中，收集更多数据会带来高昂的成本，或者根本不可行。通过将模型的训练及准确性验证看作是训练数据集大小的函数，并绘制其图像，我们可以很容易看出模型是面临高方差还是高偏差的问题，以及收集更多的数据是否有助于解决问题。在讲解如何通过scikit-learn绘制学习曲线之前，我们先通过下图来讨论一下模型常见的两个问题：



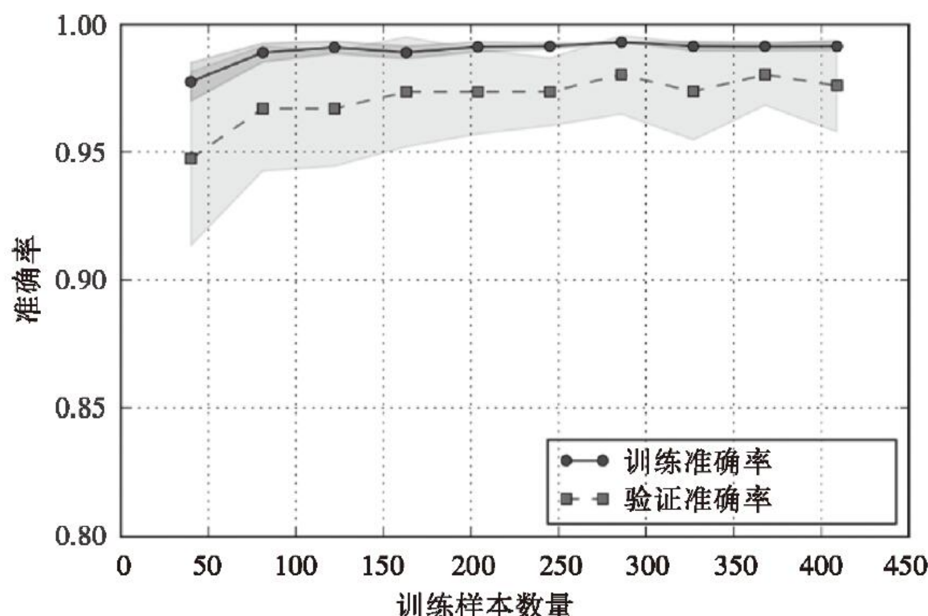
左上方图像显示的是一个高偏差模型。此模型的训练准确率和交叉验证准确率都很低，这表明此模型未能很好地拟合数据。解决此问题的常用方法是增加模型中参数的数量，例如，收集或构建额外特征，或者降低类似于SVM和逻辑斯谛回归器等模型的正则化程度。右上方图像中的模型面临高方差的问题，表明训练准确度与交叉验证准确度之间有很大差距。针对此类过拟合问题，我们可以收集更多的训练数据或者降低模型的复杂度，如增加正则化的参数；对于不适合正则化的模型，也可以通过第4章介绍的特征选择，或者第5章中的特征提取来降低特征的数量。需要注意：收集更多的训练数据可以降低模型过拟合的概率。不过该方法并不适用于所有问题，例如：训练数据中噪声数据较多，或者模型本身已经接近

于最优。

在下一小节中，我们将看到如何使用验证曲线来解决模型中存在的问题，不过先看一下如何使用scikit-learn中的学习曲线函数评估模型：

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.learning_curve import learning_curve
>>> pipe_lr = Pipeline([
...             ('scl', StandardScaler()),
...             ('clf', LogisticRegression(
...                          penalty='l2', random_state=0))])
>>> train_sizes, train_scores, test_scores =\
...         learning_curve(estimator=pipe_lr,
...                        X=X_train,
...                        y=y_train,
...                        train_sizes=np.linspace(0.1, 1.0, 10),
...                        cv=10,
...                        n_jobs=1)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(train_sizes, train_mean,
...          color='blue', marker='o',
...          markersize=5,
...          label='training accuracy')
>>> plt.fill_between(train_sizes,
...                  train_mean + train_std,
...                  train_mean - train_std,
...                  alpha=0.15, color='blue')
>>> plt.plot(train_sizes, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='validation accuracy')
>>> plt.fill_between(train_sizes,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Number of training samples')
>>> plt.ylabel('Accuracy')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

执行上述代码，可以得到如下学习曲线：



通过learning_curve函数的train_size参数，我们可以控制用于生成学习曲线的样本的绝对或相对数量。在此，通过设置train_sizes＝np.linspace（0.1,1.0,10）来使用训练数据集上等距间隔的10个样本。默认情况下，learning_curve函数使用分层k折交叉验证来计算交叉验证的准确性，通过cv参数将k的值设置为10。然后，我们可以简单地通过不同规模训练集上返回的交叉验证和测试评分来计算平均准确率，并且，我们使用matplotlib的plot函数绘制出准确率图像。此外，在绘制图像时，我们通过fill_between函数加入了平均准确率标准差的信息，用以表示评价结果的方差。

通过前面学习曲线图像可见，模型在测试数据集上表现良好。但是，在训练准确率曲线与交叉验证准确率之间，存在着相对较小差距，这意味着模型对训练数据有轻微的过拟合。

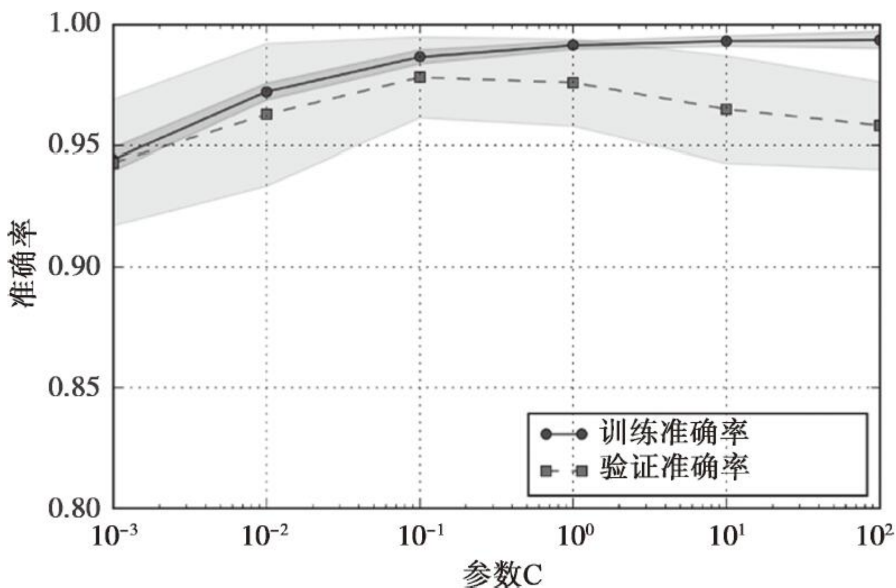
6.3.2 通过验证曲线来判定过拟合与欠拟合

验证曲线是一种通过定位过拟合或欠拟合等诸多问题所在，来帮助提高模型性能的有效工具。验证曲线与学习曲线相似，不过绘制的不是样本大小与训练准确率、测试准确率之间的函数关系，而是准确率与模型参数之间的关系，例如，逻辑斯谛回归模型中的正则化参数C。我们继续看下如何使用scikit-learn来绘制验证曲线：

```
>>> from sklearn.learning_curve import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
...                 estimator=pipe_lr,
...                  X=X_train,
...                  y=y_train,
...                  param_name='clf__C',
...                  param_range=param_range,
...                  cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(param_range, train_mean,
...          color='blue', marker='o',
...          markersize=5,
...          label='training accuracy')
>>> plt.fill_between(param_range, train_mean + train_std,
```

```
...                     train_mean - train_std, alpha=0.15,
...                     color='blue')
>>> plt.plot(param_range, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='validation accuracy')
>>> plt.fill_between(param_range,
...                     test_mean + test_std,
...                     test_mean - test_std,
...                     alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Parameter C')
>>> plt.ylabel('Accuracy')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

使用上述代码，我们可得到参数C的验证曲线图像：



与learning_curve函数类似，如果我们使用的是分类算法，则validation_curve函数默认使用分层k折交叉验证来评价模型的性能。在

validation_curve函数内，我们可以指定想要验证的参数。在本例中，需要验证的是参数C，即定义在scikit-learn流水线中的逻辑斯谛回归分类器的正则化参数，我们将其记为'clf_C'，并通过param_range参数来设定其值的范围。与上一节的学习曲线类似，我们绘制了平均训练准确率、交叉验证准确率及对应的标准差。

虽然不同C值之间准确率的差异非常小，但我们可以看到，如果加大正则化强度（较小的C值），会导致模型轻微的欠拟合；如果增加C的值，这意味着降低正则化的强度，因此模型会趋向于过拟合。在本例中，最优点在C = 0.1附近。

6.4 使用网格搜索调优机器学习模型

在机器学习中，有两类参数：通过训练数据学习得到的参数，如逻辑斯谛回归中的回归系数；以及学习算法中需要单独进行优化的参数。后者即为调优参数，也称为超参，对模型来说，就如逻辑斯谛回归中的正则化系数，或者决策树中的深度参数。

在上一节中，我们使用验证曲线通过调优超参提高模型的性能。本节中，我们将学习一种功能强大的超参数优化技巧：网格搜索（grid search），它通过寻找最优的超参值的组合以进一步提高模型的性能。

6.4.1 使用网络搜索调优超参

网格搜索法非常简单，它通过对我们指定的不同超参列表进行暴力穷举搜索，并计算评估每个组合对模型性能的影响，以获得参数的最优组合。

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.svm import SVC
>>> pipe_svc = Pipeline([('scl', StandardScaler()),
...                      ('clf', SVC(random_state=1))])
>>> param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{'clf__C': param_range,
...                'clf__kernel': ['linear']},
...               {'clf__C': param_range,
...                'clf__gamma': param_range,
...                'clf__kernel': ['rbf']}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=10,
...                   n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.978021978022
>>> print(gs.best_params_)
{'clf__C': 0.1, 'clf__kernel': 'linear'}
```

使用上述代码，我们初始化了一个sklearn.grid_search模块下的GridSearchCV对象，用于对支持向量机流水线的训练与调优。我们将GridSearchCV的param_grid参数以字典的方式定义为待调优的参数。对线性SVM来说，我们只需调优正则化参数（C）；对基于RBF的核SVM来说，我们同时需要调优C和gamma参数。请注意此处的gamma是针对核SVM特别定义的。在训练数据集上完成网格搜索后，可以通过best_scroe_属性得到最优模型的性能评分，具体参数信息可通过best_params_属性得到。在本例中，当'cfl_C' = 0.1时，线性SVM模型可得到的最优k折交叉验

证准确率为97.8%。

最后，我们将使用独立的测试数据集，通过GridSearchCV对象的best_estimator_属性对最优模型进行性能评估：

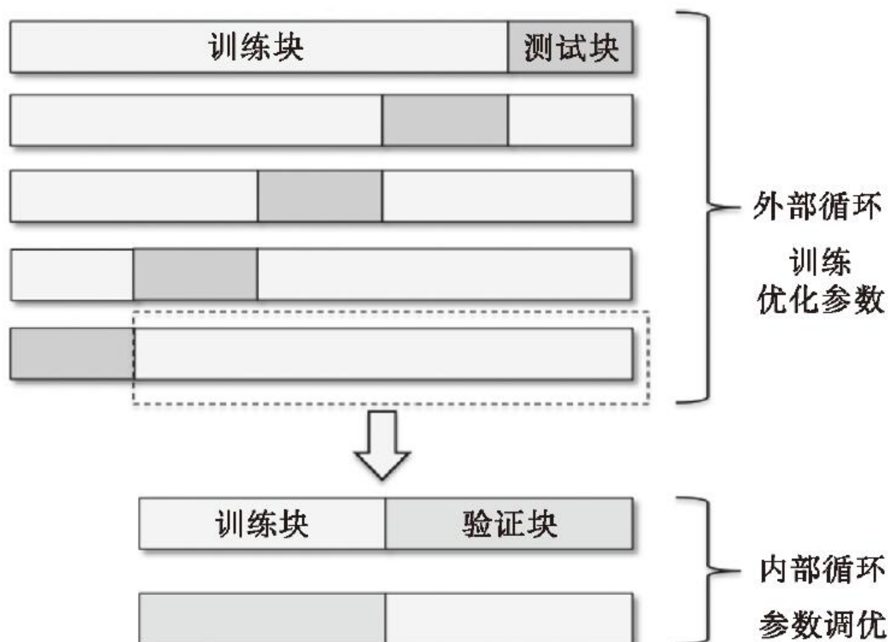
```
>>> clf = gs.best_estimator_
>>> clf.fit(X_train, y_train)
>>> print('Test accuracy: %.3f' % clf.score(X_test, y_test))
Test accuracy: 0.965
```

虽然网格搜索是寻找最优参数集合的一种功能强大的方法，但评估所有参数组合的计算成本也是相当昂贵的。使用scikit-learn抽取不同参数组合的另一种方法就是随机搜索（randomized search）。借助于scikit-learn中的RandomizedSearchCV类，我们可以以特定的代价从抽样分布中抽取出随机的参数组合。关于此方法的更详细细节及其示例请访问链接：http://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization。

6.4.2 通过嵌套交叉验证选择算法

在上一节中我们看到，结合网格搜索进行k折交叉验证，通过超参数值的改动对机器学习模型进行调优，这是一种有效提高机器学习模型性能的方法。如果要在不同机器学习算法中做出选择，则推荐另外一种方法——嵌套交叉验证，在一项对误差估计的偏差情形研究中，Varma和Simon给出了如下结论：使用嵌套交叉验证，估计的真实误差与在测试集上得到的结果几乎没有差距[1]。

在嵌套交叉验证的外围循环中，我们将数据划分为训练块及测试块；而在用于模型选择的内部循环中，我们则基于这些训练块使用k折交叉验证。在完成模型的选择后，测试块用于模型性能的评估。下图通过5个外围模块及2个内部模块解释了嵌套交叉验证的概念，这适用于计算性能要求比较高的大规模数据集。这种特殊类型的嵌套交叉验证也称为5×2交叉验证（5×2 cross-validation）：



借助于scikit-learn，我们可以通过如下方式使用嵌套交叉验证：

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=10,
...                   n_jobs=-1)
>>> scores = cross_val_score(gs, X, y, scoring='accuracy', cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...               np.mean(scores), np.std(scores)))
CV accuracy: 0.978 +/- 0.012
```

代码返回的交叉验证准确率平均值对模型超参调优的预期值给出了很好的估计，且使用该值优化过的模型能够预测未知数据。例如，我们可以使用嵌套交叉验证方法比较SVM模型与简单的决策树分类器；为了简单起见，我们只调优树的深度参数：

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...         estimator=DecisionTreeClassifier(random_state=0),

...         param_grid=[
...               {'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
...         scoring='accuracy',
...         cv=5)
>>> scores = cross_val_score(gs,
...                          X_train,
...                          y_train,
...                          scoring='accuracy',
...                          cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (
...                   np.mean(scores), np.std(scores)))
CV accuracy: 0.908 +/- 0.045
```

在此可见，嵌套交叉验证对SVM模型性能的评分（97.8%）远高于决策树的（90.8%）。由此，可以预期：SVM是用于对此数据集未知数据进行分类的一个好的选择。

[1] A.Varma and R.Simon.Bias in Error Estimation When Using Cross-validation for Model Selection.BMC bioinformatics，7(1):91,2006.

6.5 了解不同的性能评价指标

在前面章节中，我们使用模型准确性对模型进行了评估，这是通常情况下有效量化模型性能的一个指标。不过还有其他几个性能指标可以用来衡量模型的相关性能，比如准确率（precision）、召回率（recall）以及F1分数（F1-score）。

6.5.1 读取混淆矩阵

在我们详细讨论不同评分标准之前，先绘制一个所谓的混淆矩阵（confusion matrix）：即展示学习算法性能的一种矩阵。混淆矩阵是一个简单的方阵，用于展示一个分类器预测结果——真正（true positive）、真负（false negative）、假正（false positive）及假负（false negative）——的数量，如下图所示：

实际类别 \ 预测类别	P	N
P	真正 (TP)	假负 (FN)
N	假正 (FP)	真负 (TN)

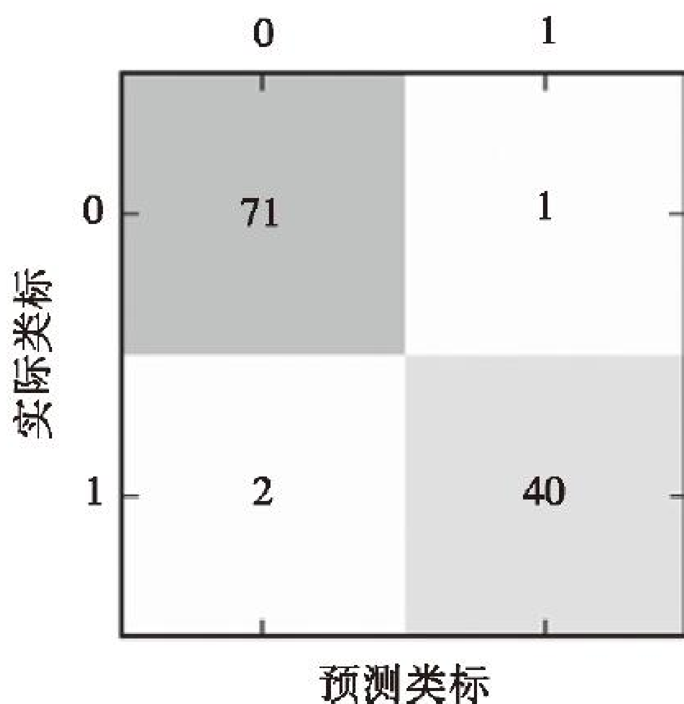
虽然这指标的数据可以通过人工比较真实类标与预测类标来获得，不过scikit-learn提供了一个方便使用的confusion_matrix函数，其使用方法如下：

```
>>> from sklearn.metrics import confusion_matrix
>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>>> print(confmat)
[[71  1]
 [ 2 40]]
```

在执行上述代码后，返回的数组提供了分类器在测试数据集上生成的不同错误信息，我们可以使用matplotlib中的matshow函数将它们表示为上图所示的混淆矩阵形式：

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i,
...                 s=confmat[i, j],
...                 va='center', ha='center')
>>> plt.xlabel('predicted label')
>>> plt.ylabel('true label')
>>> plt.show()
```

下图所示的混淆矩阵使得预测结果更易于解释：



在本例中，假定类别1（恶性）为正类，模型正确地预测了71个属于类别0的样本（真负），以及40个属于类别1的样本（真正）。不过，我们的模型也错误地将两个属于类别0的样本划分到了类别1（假负），另外还将一个恶性肿瘤误判为良性的（假正）。在下一节中，我们将使用这些信息计算不同的误差度量。

6.5.2 优化分类模型的准确率和召回率

预测误差（error，ERR）和准确率（accuracy，ACC）都提供了误分类样本数量的相关信息。误差可以理解为预测错误样本的数量与所有被预测样本数量的比值，而准确率计算方法则是正确预测样本的数量与所有被预测样本数量的比值：

$$ERR=\frac{FP+FN}{FP+FN+TP+TN}$$

预测准确率也可以通过误差直接计算：

$$ACC=\frac{TP+TN}{FP+FN+TP+TN}=1-ERR$$

对于类别数量不均衡的分类问题来说，真正率（TPR）与假正率（FPR）是非常有用的性能指标：

$$FPR=\frac{FP}{N}=\frac{FP}{FP+TN}$$

$$TPR=\frac{TP}{P}=\frac{TP}{FN+TP}$$

以肿瘤诊断为例，我们更为关注的是正确检测出恶性肿瘤，使得病人得到恰当治疗。然而，降低良性肿瘤（假正）错误地划分为恶性肿瘤事例的数量固然重要，但对患者来说影响并不大。与FPR相反，真正率提供了有关正确识别出来的恶性肿瘤样本（或相关样本）的有用信息。

准确率（precision，PRE）和召回率（recall，REC）是与真正率、真

负率相关的性能评价指标，实际上，召回率与真正率含义相同：

$$PRE=\frac{TP}{TP+FP}$$

$$REC=TPR=\frac{TP}{P}=\frac{TP}{FN+TP}$$

在实践中，常采用准确率与召回率的组合，称为F1分数：

$$F1=2\frac{PRE\times REC}{PRE+REC}$$

所有这些评分指标均以在scikit-learn中实现，可以从sklearn.metric模块中导入使用，示例代码如下：

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics  import recall_score, f1_score
>>> print('Precision: %.3f' % precision_score(
...                 y_true=y_test, y_pred=y_pred))
Precision: 0.976
>>> print('Recall: %.3f' % recall_score(
...                 y_true=y_test, y_pred=y_pred))
Recall: 0.952
>>> print('F1: %.3f' % f1_score(
...                 y_true=y_test, y_pred=y_pred))
F1: 0.964
```

此外，通过评分参数，我们还可以在GridSerch中使用包括准确率在内的其他多种不同评分标准。可通过网址http://scikit-learn.org/stable/modules/model_evaluation.html了解评分参数可使用的所有不同值的列表。

请记住scikit-learn中将正类类标标识为1。如果我们想指定一个不同的正类类标，可通过make_scorer函数来构建我们自己的评分，那样我们就可以将其以参数的形式提供给GridSearchCV：

```
>>> from sklearn.metrics import make_scorer, f1_score
>>> scorer = make_scorer(f1_score, pos_label=0)
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring=scorer,
...                   cv=10)
```

6.5.3 绘制ROC曲线

受试者工作特征曲线（receiver operator characteristic，ROC）是基于模型假正率和真正率等性能指标进行分类模型选择的有用工具，假正率和真正率可以通过移动分类器的分类阈值来计算。ROC的对角线可以理解为随机猜测，如果分类器性能曲线在对角线以下，那么其性能就比随机猜测还差。对于完美的分类器来说，其真正率为1，假正率为0，这时的ROC曲线即为横轴0与纵轴1组成的折线。基于ROC曲线，我们就可以计算所谓的ROC线下区域（area under the curve，AUC），用来刻画分类模型的性能。

与ROC曲线类似，我们也可以计算不同概率阈值下一个分类器的准确率-召回率曲线。绘制此准确率-召回率曲线的方法也在scikit-learn中得以实现，其文档请参阅链接：http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html。

在下面的代码中，我们只使用了威斯康星乳腺癌数据集中的两个特征来判定肿瘤是良性还是恶性，并绘制了相应的ROC曲线。虽然再次使用了前面定义的逻辑斯谛回归流水线，但是为了使ROC曲线视觉上更加生动，我们对分类器的设置比过去更具有挑战性。出于相同的考虑，我们还将StratifiedKFold验证器中的分块数量减少为3。代码如下：

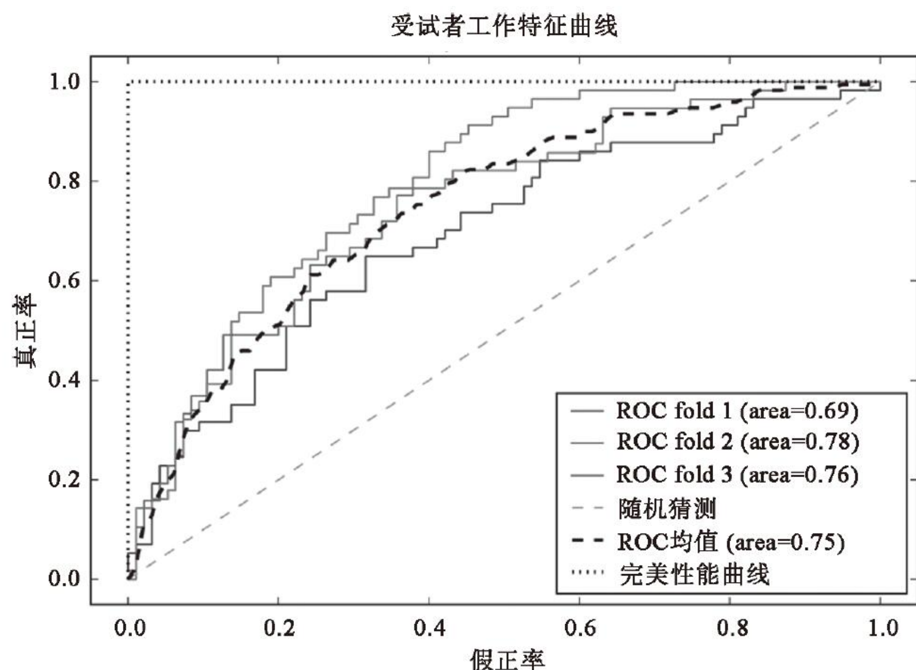
```
>>> from sklearn.metrics import roc_curve, auc
>>> from scipy import interp
>>> X_train2 = X_train[:, [4, 14]]
>>> cv = StratifiedKFold(y_train,
...                      n_folds=3,
...                      random_state=1)
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []

>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(X_train2[train],
>>> y_train[train]).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
...                                      probas[:, 1],
...                                      pos_label=1)
...     mean_tpr += interp(mean_fpr, fpr, tpr)
...     mean_tpr[0] = 0.0
...     roc_auc = auc(fpr, tpr)
...     plt.plot(fpr,
...              tpr,
...              lw=1,
...              label='ROC fold %d (area = %0.2f)'
...                    % (i+1, roc_auc))
>>> plt.plot([0, 1],
...          [0, 1],
...          linestyle='--',
...          color=(0.6, 0.6, 0.6),
...          label='random guessing')
>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...          label='mean ROC (area = %0.2f)' % mean_auc, lw=2)
>>> plt.plot([0, 0, 1],
...          [0, 1, 1],
...          lw=2,
...          linestyle=':',
...          color='black',

...          label='perfect performance')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('false positive rate')
>>> plt.ylabel('true positive rate')
>>> plt.title('Receiver Operator Characteristic')
>>> plt.legend(loc="lower right")
>>> plt.show()
```

前面的示例代码使用了scikit-learns中我们已经熟悉的StratifiedKFold

类，并且在pipe_lr流水线的每次迭代中都使用了sklearn.metrics模块中的roc_curve函数，以计算Logistic-Regresssion分类器的ROC性能。此外，我们通过SciPy中的interp函数利用三个块数据对ROC曲线的内插均值进行了计算，并使用auc函数计算了低于ROC曲线区域的面积。最终的ROC曲线表明不同的块之间存在一定的方差，且平均ROC AUC（0.75）位于最理想情况（1.0）与随机猜测（0.5）之间。



如果我们仅对ROC AUC的得分感兴趣，也可以直接从sklearn.metircs子模块中导入roc_auc_score函数。分类器在只有两个特征的训练集上完成拟合后，使用如下代码计算分类器在单独测试集上的ROC AUC得分：

```
>>> pipe_svc = pipe_svc.fit(X_train2, y_train)
>>> y_pred2 = pipe_svc.predict(X_test[:, [4, 14]])
>>> from sklearn.metrics import roc_auc_score
>>> from sklearn.metrics import accuracy_score
>>> print('ROC AUC: %.3f' % roc_auc_score(
...         y_true=y_test, y_score=y_pred2))
ROC AUC: 0.671

>>> print('Accuracy: %.3f' % accuracy_score(
...         y_true=y_test, y_pred=y_pred2))
Accuracy: 0.728
```

通过ROC AUC得到的分类器性能可以让我们进一步洞悉分类器在类别不均衡样本集合上的性能。然而，既然准确率评分可以解释为ROC曲线上某个单点处的值，A.P.Bradley认为ROC AUC与准确率矩阵之间是相互一致的[1]。

[1] A.P.Bradley.The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms.Pattern recognition，30(7): 1145-1159，1997.

6.5.4　多类别分类的评价标准

本节中讨论的评分标准都是基于二类别分类系统的。不过，scikit-learn实现了macro（宏）及micro（微）均值方法，旨在通过一对多（One vs All，OvA）的方式将评分标准扩展到了多类别分类问题。微均值是通过系统的真正、真负、假正，以及假负来计算的。例如，k类分类系统的准确率评分的微均值可按如下公式进行计算：

$$PRE_{micro}=\frac{TP_1+\cdots+TP_k}{TP_1+\cdots+TP_k+FP_1+\cdots+FP_k}$$

宏均值仅计算不同系统的平均分值：

$$PRE_{macro}=\frac{PRE_1+\cdots+PRE_k}{k}$$

当我们等同看待每个实例或每次预测时，微均值是有用的，而宏均值则是我们等同看待各个类别，将其用于评估分类器针对最频繁类标（即样本数量最多的类）的整体性能。

如果我们使用别二类别分类性能指标来衡量scikit-learn中的多类别分类模型，会默认使用一个归一化项或者是宏均值的一个加权变种。计算加权宏均值时，各类别以类内实例的数量作为评分的权值。当数据中类中样本分布不均衡时，也就是类标数量不一致时，采用加权宏均值比较有效。

由于加权宏均值是scikit-learn中多类别问题的默认值，我们可以通过sklearn.metrics模块导入其他不同的评分函数，如precision_score或make_scorer函数等，并利用函数内置的avarage参数定义平均方法：

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                          pos_label=1,
...                          greater_is_better=True,
...                          average='micro')
```

本章小结

在本章的开始，我们讨论了通过便捷的流水线模型串联不同的数据转换技术与分类器，以帮助我们更高效地训练与评估机器学习模型。进而我们使用流水线进行k折交叉验证，k折交叉验证是模型选择及评估的一种基本技术。使用k折交叉验证，我们绘制了学习曲线和验证曲线以诊断学习算法中过拟合与欠拟合等常见问题。使用网格搜索，进一步对模型进行微调。最后我们学习了混淆矩阵以及各种不同的性能评价指标，在针对特定问题需要进一步优化模型时，这些指标可能是非常有用的。到目前为止，我们已经具备了使用监督机器学习模型来成功构建分类器的基本技能。

在下一章，我们将学习算法集成方法，它使得我们可以通过混合多个模型与分类算法进一步提高机器学习系统的性能。

第7章　集成学习——组合不同的模型

在上一章中，我们专注于不同分类模型的调优和评估等任务的实战操作。在本章中，我们将在这些技术的基础上探索一种新的方法，构建一组分类器的集合，使得整体分类效果优于其中任意一个单独的分类器。我们将学到：

·基于多数投票的预测

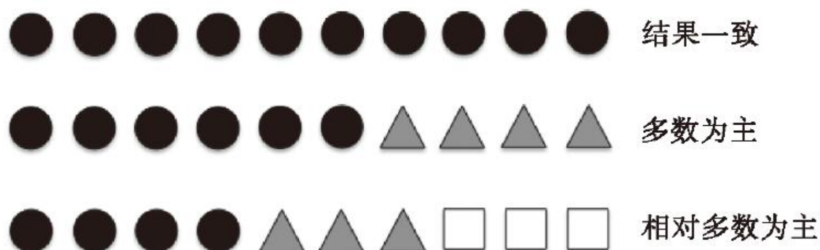
·通过对训练数据集的重复抽样和随机组合降低模型的过拟合

·通过弱学习机在误分类数据上的学习构建性能更好的模型

7.1 集成学习

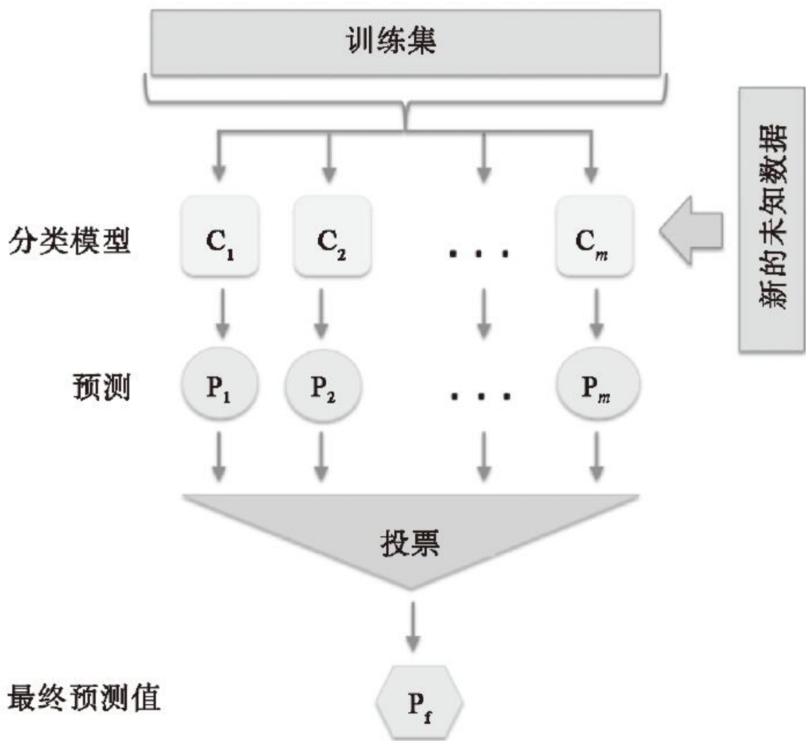
集成方法（ensemble method）的目标是：将不同的分类器组合成为一个元分类器，与包含于其中的单个分类器相比，元分类器具有更好的泛化性能。例如：假设我们收集到了10位专家的预测结果，集成方法允许我们以一定策略将这10位专家的预测进行组合，与每位专家单独的预测相比，它具有更好的准确性和鲁棒性。本章我们将介绍几种不同的分类器集成方法。在本节，我们先介绍集成方法是如何工作的，以及它们为何能具备良好的泛化性能。

本章聚焦几种最流行的集成方法，它们都使用了多数投票（majority voting）原则。多数投票原则是指将大多数分类器预测的结果作为最终类标，也就是说，将得票率超过50%的结果作为类标。严格来说，多数投票仅用于二类别分类情形。不过，很容易将多数投票原则推广到多类别分类，也称作简单多数票法（plurality voting）。在此，我们选择得票最多的类标。下图解释了集成10个分类器时，多数及简单多数票法表决的概念，其中每个单独的符号（三角形、正方形和圆）分别代表一个类标：



基于训练集，我们首先训练m个不同的成员分类器（$C_1,\ldots,C_m$）。在

多数投票原则下，可集成不同的分类算法，如决策树、支持向量机、逻辑斯谛回归等。此外，我们也可以使用相同的成员分类算法拟合不同的训练子集。这种方法典型的例子就是随机森林算法，它组合了不同的决策树分类器。下图解释了使用多数投票原则的通用集成方法的概念：



想要通过简单的多数投票原则对类标进行预测，我们要汇总所有分类器C_j的预测类标，并选出得票率最高的类标$\hat{y}$：

$$\hat{y}=mode\{C_1(\boldsymbol{x}),C_2(\boldsymbol{x}),\cdots,C_m(\boldsymbol{x})\}$$

例如，在二类别分类中，假定class1 = -1、class2 = ＋1，我们可以将多数投票预测表示为：

$$C(\boldsymbol{x})=sign\left[\sum_{j}^{m}C_j(\boldsymbol{x})\right]=\begin{cases}1 & 若\sum_{i}C_j(\boldsymbol{x})\geqslant 0\\-1 & 其他\end{cases}$$

为了说明集成方法的效果为何好于单个成员分类器，我们借用下组合学中的概念。接下来的例子中，我们假定二类别分类中的n个成员分类器都有相同的出错率ε。此外，假定每个分类器都是独立的，且出错率之间是不相关的。基于这些假设，我们可以将成员分类器集成后的出错概率简单地表示为二项分布的概率密度函数：

$$P(y\geqslant k)=\sum_{k}^{n}\left\langle\begin{matrix}n\\k\end{matrix}\right\rangle\varepsilon^k(1-\varepsilon)^{n-k}=\varepsilon_{ensemble}$$

其中，$\left\langle\begin{matrix}n\\k\end{matrix}\right\rangle$是n选k组合的二项式系数。换句话说，我们计算此集成分类器预测结果出错的概率。再来看一个更具体的例子：假定使用11个分类器（n = 11），单个分类器出错率为0.25（ε = 0.25）：

$$P(y\geqslant k)=\sum_{k=6}^{n}\left\langle\begin{matrix}11\\k\end{matrix}\right\rangle 0.25^k(1-\varepsilon)^{11-k}=0.034$$

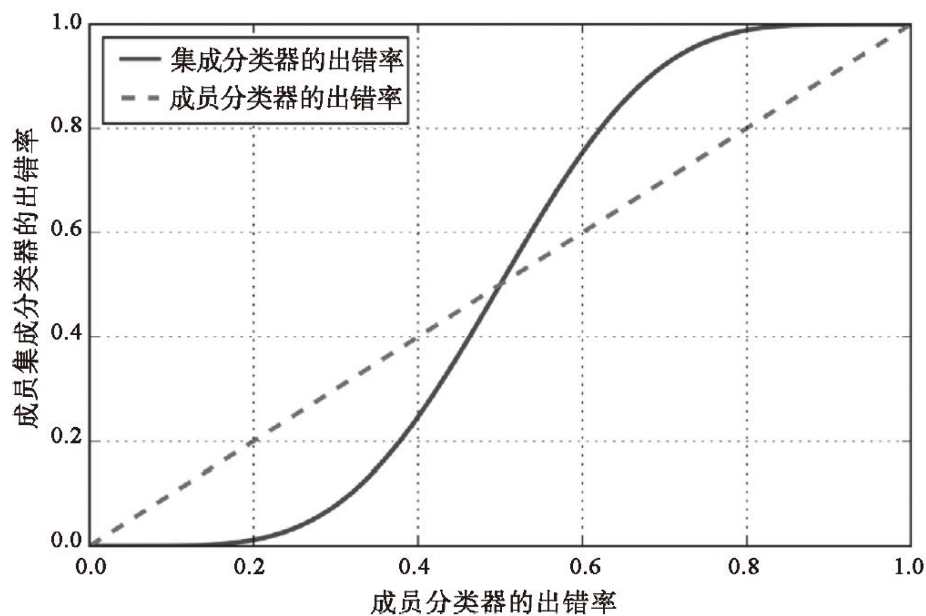
正如我们所见，在满足所有假设的条件下，集成后的出错率（0.034）远远小于单个分类器的出错率（0.25）。请注意，在此演示示例中，当集成分类中分类器个数n为偶数时，若预测结果为五五分，我们则将其以错误对待，不过仅有一半的可能会出现这种情况。为了比较成员分类器在不同出错率的情况下与集成分类器出错率的差异，我们用Python实现其概率密度函数：

```
>>> from scipy.misc import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = math.ceil(n_classifier / 2.0)
...     probs = [comb(n_classifier, k) *
...              error**k *
...              (1-error)**(n_classifier - k)
...              for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.034327507019042969
```

实现ensemble_error函数后，在成员分类器出错率介于0.0到1.0范围内，我们可以计算对应集成分类器的出错率，并以函数曲线的形式绘制出它们之间的关系：

```
>>> import numpy as np
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...               for error in error_range]
>>> import matplotlib.pyplot as plt
>>> plt.plot(error_range, ens_errors,
...          label='Ensemble error',
...          linewidth=2)
>>> plt.plot(error_range, error_range,
...          linestyle='--', label='Base error',
...          linewidth=2)
>>> plt.xlabel('Base error')
>>> plt.ylabel('Base/Ensemble error')
>>> plt.legend(loc='upper left')
>>> plt.grid()
>>> plt.show()
```

从结果图像中可见：当成员分类器出错率低于随机猜测时（$\varepsilon < 0.5$），集成分类器的出错率要低于单个分类器。请注意：y轴同时标识了成员分类器的出错率（虚线）和集成分类器的出错率（实线）：


集成分类器的出错率
成员分类器的出错率
成员集成分类器的出错率
成员分类器的出错率
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

7.2 实现一个简单的多数投票分类器

通过上一小节的介绍，我们已经对集成学习有了初步的认识，现在来做一个热身练习：基于多数投票原则，使用Python实现一个简单的集成分类器。虽然下述算法可通过简单多数投票推广到多分类器的情形，不过为了简单起见，我们仍旧使用更常出现在文献中的术语：多数投票（majority voting）。

集成算法允许我们使用单独的权重对不同分类算法进行组合。我们的目标是构建一个更加强大的元分类器，以在特定的数据集上平衡单个分类器的弱点。通过更严格的数学概念，可以将加权多数投票记为：

$$\hat{y}=\arg\max_{i}\sum_{j=1}^{m}w_j\chi_A(C_j(\boldsymbol{x})=i)$$

其中，w_j是成员分类器C_j对应的权重，$\hat{y}$为集成分类器的预测类标，χ_A（希腊字母chi）为特征函数[C_j（x）= i∈A]，A为类标的集合。如果权重均等，可以将此公式简化为：

$$\hat{y}=mode\{C_1(\boldsymbol{x}),C_2(\boldsymbol{x}),\cdots,C_m(\boldsymbol{x})\}$$

为了更好地理解权重在这里的含义，我们来看几个更加具体的例子。假定有三个成员分类器C_j（j∈{1,2,3}），分别用它们来预测样本x的类标。其中两个成员分类器的预测结果为类别0，而另外一个分类器C_3的预测结果为类别1。如果我们平等地看待每个成员分类器，基于多数投票原则，最终的预测结果应该是样本属于类别0：

$$C_1(x)\rightarrow 0,\ C_2(x)\rightarrow 0,\ C_3(x)\rightarrow 1$$
$$\hat{y}=mode\{0,0,1\}=0$$

我们将权重0.6赋给C_3，而C_1和C_2均为0.2，有：

$$\hat{y}=\arg\max_i \sum_{j=1}^{m} w_j \chi_A(C_j(\boldsymbol{x})=i)$$
$$=\arg\max_i [0.2\times i_0+0.2\times i_0+0.6\times i_1]=1$$

更直观地，由于$3\times0.2=0.6$，可以认为分类器C_3的一次预测权重相当于分类器C_1或C_2的三次预测权重之和，我们可以写作：

$$\hat{y}=mode\{0,0,1,1,1\}=1$$

为了使用Python代码实现加权多数投票，可以使用NumPy中的argmax和bincount函数：

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...             weights=[0.2, 0.2, 0.6]))
1
```

我们在第3章中曾讨论过，通过predict_proba方法，scikit-learn中的分类器可以返回样本属于预测类标的概率。如果集成分类器事先得到良好的修正，那么在多数投票中使用预测类别的概率来替代类标会非常有用。使用类别概率进行预测的多数投票修改版本可记为：

$$\hat{y}=\arg\max_i \sum_{j=1}^{m} w_j p_{ij}$$

其中，p_{ij}是第j个分类器预测样本类标为i的概率。

继续前面的例子，假设我们面临一个二类别分类问题，其类标为$i\in\{0,1\}$，下面集成这三个分类器C_j（$j\in\{1,2,3\}$）。假定分类器C_j针对某一

样本x按照如下概率返回类标的预测结果：

$$C_1(x) \rightarrow [0.9, 0.1], C_2(x) \rightarrow [0.8, 0.2], C_3(x) \rightarrow [0.4, 0.6]$$

我们可以按照如下方式计算所属类别的概率：

$$p(i_0|\boldsymbol{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$
$$p(i_1|\boldsymbol{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.06 = 0.42$$
$$\hat{y} = \arg\max_i [p(i_0|\boldsymbol{x}), p(i_1|\boldsymbol{x})] = 0$$

为实现基于类别预测概率的加权多数投票，我们可以再次使用NumPy中的numPy.average和np.argmax方法：

```
>>> ex = np.array([[0.9, 0.1],
...                [0.8, 0.2],
...                [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([ 0.58,  0.42])
>>> np.argmax(p)
0
```

综合上述内容，我们使用Python来实现MajorityVoteClassifier类：

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.externals import six
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator


class MajorityVoteClassifier(BaseEstimator,
                             ClassifierMixin):
    """ A majority vote ensemble classifier

    Parameters
    ----------
    classifiers : array-like, shape = [n_classifiers]
      Different classifiers for the ensemble

    vote : str, {'classlabel', 'probability'}
      Default: 'classlabel'
      If 'classlabel' the prediction is based on
      the argmax of class labels. Else if
      'probability', the argmax of the sum of
      probabilities is used to predict the class label
      (recommended for calibrated classifiers).

    weights : array-like, shape = [n_classifiers]
      Optional, default: None
      If a list of `int` or `float` values are
  provided, the classifiers are weighted by
  importance; Uses uniform weights if `weights=None`.

"""
def __init__(self, classifiers,
             vote='classlabel', weights=None):
```

```
        self.classifiers = classifiers
        self.named_classifiers = {key: value for
                                  key, value in
                                  _name_estimators(classifiers)}
        self.vote = vote
        self.weights = weights

    def fit(self, X, y):
        """ Fit classifiers.

        Parameters
        ----------
        X : {array-like, sparse matrix},
            shape = [n_samples, n_features]
            Matrix of training samples.

        y : array-like, shape = [n_samples]
            Vector of target class labels.

        Returns
        -------
        self : object

        """
        # Use LabelEncoder to ensure class labels start
        # with 0, which is important for np.argmax
        # call in self.predict
        self.lablenc_ = LabelEncoder()
        self.lablenc_.fit(y)
        self.classes_ = self.lablenc_.classes_
        self.classifiers_ = []
        for clf in self.classifiers:
            fitted_clf = clone(clf).fit(X,
                              self.lablenc_.transform(y))
            self.classifiers_.append(fitted_clf)
        return self
```

为了帮助读者更好地理解各部分内容，我在代码中加入了大量的注释。不过，在实现其他的方法之前，我们稍做中断，先快速地讨论一下看似让人眼花缭乱的代码。我们使用两个基类BaseEstimator和ClassifierMixin获取某些基本方法，包括设定分类器参数的set_params和返回参数的get_params方法，以及用于计算预测准确率的score方法。此外请注意，我们导入six包从而使得MajorityVoteClassifier与Python 2.7兼容。

接下来我们加入predict方法：如果使用参数vote＝'classlabel'初始化MajorityVote-Classifier对象，我们就可通过多数投票来预测类标，相反，如果使用参数vote＝'probability'初始化集成分类器，则可基于类别成员的概率进行类标预测。此外，我们还将加入predict_proba方法来返回平均概率，这在计算受试工作者特征线下区域（Receiver Operator Characteristic area under the curve，ROC AUC）时需要用到。

```
def predict(self, X):
    """ Predict class labels for X.

    Parameters
    ----------
```

```
        X : {array-like, sparse matrix},
            Shape = [n_samples, n_features]
            Matrix of training samples.

        Returns
        ----------
        maj_vote : array-like, shape = [n_samples]
            Predicted class labels.

        """
        if self.vote == 'probability':
            maj_vote = np.argmax(self.predict_proba(X),
                                 axis=1)
        else:  # 'classlabel' vote

            #  Collect results from clf.predict calls
            predictions = np.asarray([clf.predict(X)
                                      for clf in
                                      self.classifiers_]).T

            maj_vote = np.apply_along_axis(
                           lambda x:
                           np.argmax(np.bincount(x,
                                         weights=self.weights)),
                           axis=1,
                           arr=predictions)
        maj_vote = self.lablenc_.inverse_transform(maj_vote)
        return maj_vote

    def predict_proba(self, X):
        """ Predict class probabilities for X.

        Parameters
        ----------
        X : {array-like, sparse matrix},
            shape = [n_samples, n_features]
            Training vectors, where n_samples is
            the number of samples and
            n_features is the number of features.

        Returns
        ----------
        avg_proba : array-like,
            shape = [n_samples, n_classes]
            Weighted average probability for
            each class per sample.

        """
        probas = np.asarray([clf.predict_proba(X)
                             for clf in self.classifiers_])
        avg_proba = np.average(probas,
                               axis=0, weights=self.weights)
        return avg_proba

    def get_params(self, deep=True):
        """ Get classifier parameter names for GridSearch"""
        if not deep:
```

```
            return super(MajorityVoteClassifier,
                         self).get_params(deep=False)
        else:
            out = self.named_classifiers.copy()
            for name, step in\
                    six.iteritems(self.named_classifiers):
                for key, value in six.iteritems(
                        step.get_params(deep=True)):
                    out['%s__%s' % (name, key)] = value
            return out
```

此外还请注意：这里还定义了我们自行修改的get_params方法，以方便使用_name_estimators函数来访问集成分类器中独立成员函数的参数。起初看起来这有点复杂，但在后续小节中，我们将使用网格搜索来调整超参，那时你会感到这样做意义非凡。

主要是出于演示的目的，我们才实现了MajorityVoteClassifier类，不过这也完成了scikit-learn中关于多数投票分类器的一个相对复杂的版本。它将出现在下一个版本（v0.17）的sklearn.ensemble.VotingClassifier类中。

基于多数投票原则组合不同的分类算法

现在我们可以将上一小节实现的MajorityVoteClassifier用于实战了。不过，先准备一个可以用于测试的数据集。既然已经熟悉了从CSV文件中读取数据的方法，我们就走捷径从scikit-learn的dataset模块中加载鸢尾花数据集。此外，为了使分类任务更具挑战，我们只选择其中的两个特征：萼片宽度和花瓣长度。虽然MajorityVoteClassifier可应用到多类别分类问题，但我们只区分两个类别的样本：Iris-Versicolor和Iris-Virginica，并绘制其ROC AUC。代码如下：

```
>>> from sklearn import datasets
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

请注意，我们可以使用scikit-learn中的predict_proba方法（如果适用）计算ROC AUC的评分。在第3章中，我们学习了如何通过逻辑斯谛回归模型来计算样本相对于各类别的归属概率；在决策树中，此概率是通过训练时为每个节点创建的频度向量（frequency vector）来计算的。此向量收集对应节点中通过类标分布计算得到各类标频率值。进而对频率进行归一化处理，使得它们的和为1。类似地，k-近邻算法中，也会收集各样本最相邻的k个邻居的类标，并返回归一化的类标频率。虽然决策树和k-近邻分类器都返回了与逻辑斯谛回归模型类似的概率值，但必须注意，这些值并非通过概率密度函数计算得到的。

下面我们将样本按照五五分划分为训练数据及测试数据：

```
>>> X_train, X_test, y_train, y_test =\

...        train_test_split(X, y,
...                         test_size=0.5,
...                         random_state=1)
```

我们使用训练数据集训练三种不同类型的分类器：逻辑斯谛回归分类器、决策树分类器及k-近邻分类器各一个，在将它们组合成集成分类器之前，我们先通过10折交叉验证看一下各个分类器在训练数据集上的性能表现：

```
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           random_state=0)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                               criterion='entropy',
...                               random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf1]])
>>> pipe3 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf3]])
>>> clf_labels = ['Logistic Regression', 'Decision Tree', 'KNN']
>>> print('10-fold cross validation:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
>>>                              X=X_train,
>>>                              y=y_train,
>>>                              cv=10,
>>>                              scoring='roc_auc')
>>>     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...           % (scores.mean(), scores.std(), label))
```

我们得到的输出显示如以下代码段，单个分类器的预测性能几乎相同：

```
10-fold cross validation:

ROC AUC: 0.92 (+/- 0.20) [Logistic Regression]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
```

读者可能会感到奇怪，为什么我们将逻辑斯谛回归和k-近邻分类器的训练作为流水线的一部分？原因在于：如我们在第3章中所述，不同于决策树，逻辑斯谛回归与k-近邻算法（使用欧几里得距离作为距离度量标准）对数据缩放不敏感。虽然鸢尾花特征都以相同的尺度（厘米）度量，不过对特征做标准化处理是一个好习惯。

激动人心的时刻到了，我们现在基于多数投票原则，在MajorityVoteClassifier中组合各成员分类器：

```
>>> mv_clf = MajorityVoteClassifier(
...                 classifiers=[pipe1, clf2, pipe3])
>>> clf_labels += ['Majority Voting']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
...                              cv=10,
...                              scoring='roc_auc')
...     print("Accuracy: %0.2f (+/- %0.2f) [%s]"
...                 % (scores.mean(), scores.std(), label))
ROC AUC: 0.92 (+/- 0.20) [Logistic Regression]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
ROC AUC: 0.97 (+/- 0.10) [Majority Voting]
```

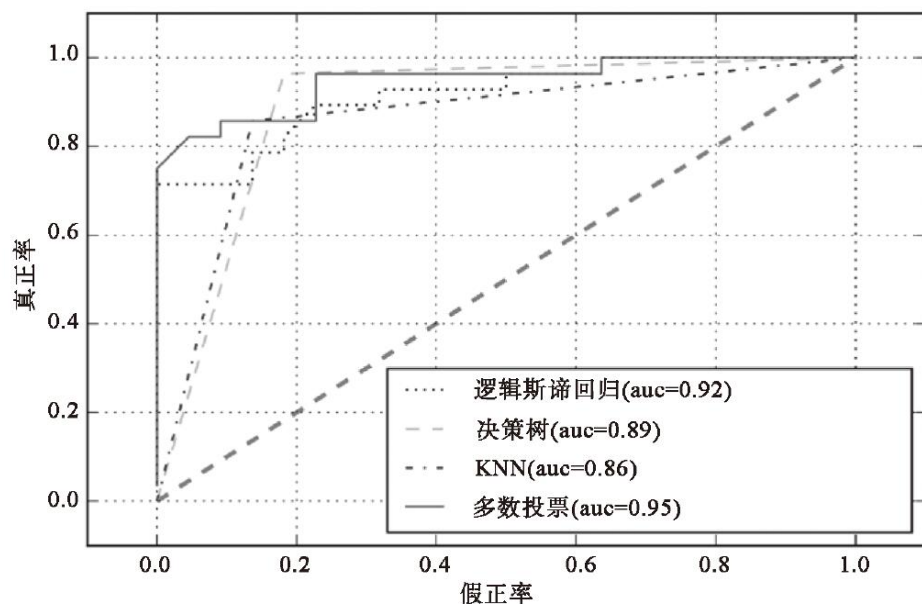
正如我们所见，以10折交叉验证作为评估标准，MajorityVotingClassifier的性能与单个成员分类器相比有着质的提高。

7.3 评估与调优集成分类器

在本节，我们将在测试数据上计算MajorityVoteClassifier类的ROC曲线，以验证其在未知数据上的泛化性能。请记住，训练数据并未应用于模型选择，使用它们的唯一目的就是对分类系统的泛化性能做出无偏差的估计。代码如下：

```
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyles = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...         in zip(all_clf, clf_labels, colors, linestyles):
...     # assuming the label of the positive class is 1
...     y_pred = clf.fit(X_train,
...                      y_train).predict_proba(X_test)[:, 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                      y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...              color=clr,
...              linestyle=ls,
...              label='%s (auc = %0.2f)' % (label, roc_auc))
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...          linestyle='--',
...          color='gray',
...          linewidth=2)
>>> plt.xlim([-0.1, 1.1])
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid()
>>> plt.xlabel('False Positive Rate')
>>> plt.ylabel('True Positive Rate')
>>> plt.show()
```

由ROC结果我们可以看到，集成分类器在测试集上表现优秀（ROC AUC = 0.95），而k-近邻分类器对于训练数据有些过拟合（训练集上的ROC AUC = 0.93，测试集上ROC AUC = 0.86）：

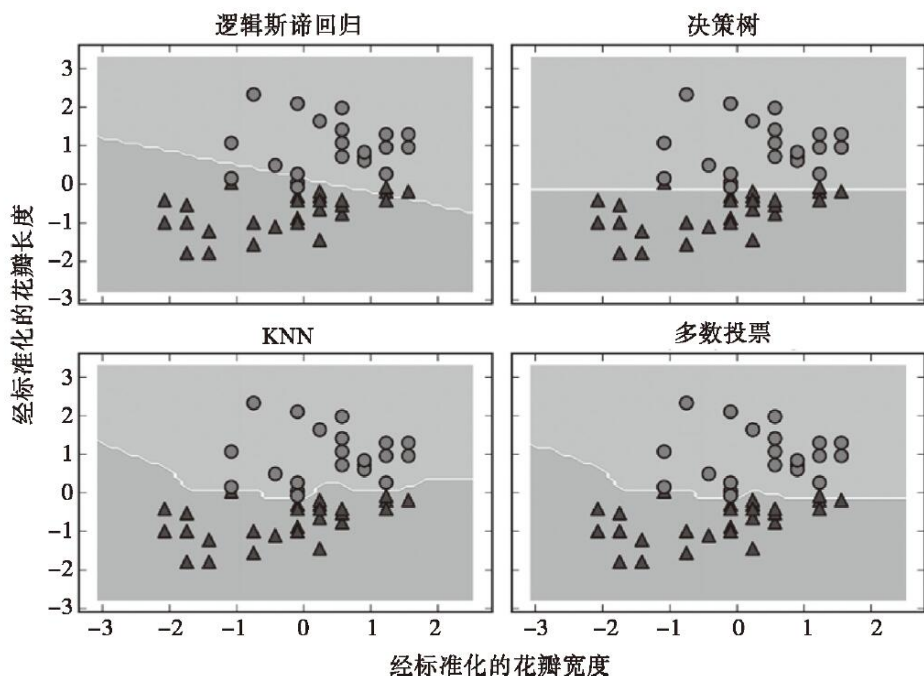


因为只使用了分类样本中的两个特征，集成分类器的决策区域到底是什么样子可能会引起我们的兴趣。由于逻辑斯谛回归和k-近邻流水线会自动对数据进行预处理，因此不必事先对训练特征进行标准化。不过出于可视化考虑，也就是在相同的度量标准上实现决策区域，我们在此对训练集做了标准化处理。代码如下：

```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>> y_max = X_train_std[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                         all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
...                                   X_train_std[y_train==0, 1],
...                                   c='blue',
...                                   marker='^',
...                                   s=50)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...                                   X_train_std[y_train==1, 1],
...                                   c='red',
...                                   marker='o',
...                                   s=50)
...     axarr[idx[0], idx[1]].set_title(tt)

>>> plt.text(-3.5, -4.5,
...          s='Sepal width [standardized]',
...          ha='center', va='center', fontsize=12)
>>> plt.text(-10.5, 4.5,
...          s='Petal length [standardized]',
...          ha='center', va='center',
...          fontsize=12, rotation=90)
>>> plt.show()
```

尽管有趣，但也如我们所预期的那样，集成分类器的决策区域看起来像是各成员分类器决策区域的混合。乍看之下，多数投票的决策区域与k-近邻分类器的很相似。不过，正如单层决策树那样，我们可以看到它在sepal width ≥ 1时正交于y轴：



在学习集成分类的成员分类器调优之前，我们调用一下get_param方法，以便对如何访问GridSearch对象内的单个参数有个基本的认识：

```
>>> mv_clf.get_params()
{'decisiontreeclassifier': DecisionTreeClassifier(class_weight=None,
criterion='entropy', max_depth=1,
             max_features=None, max_leaf_nodes=None, min_samples_
leaf=1,
             min_samples_split=2, min_weight_fraction_leaf=0.0,
             random_state=0, splitter='best'),
 'decisiontreeclassifier__class_weight': None,
 'decisiontreeclassifier__criterion': 'entropy',
 [...]
 'decisiontreeclassifier__random_state': 0,
 'decisiontreeclassifier__splitter': 'best',
 'pipeline-1': Pipeline(steps=[('sc', StandardScaler(copy=True, with_
mean=True, with_std=True)), ('clf', LogisticRegression(C=0.001, class_
weight=None, dual=False, fit_intercept=True,
           intercept_scaling=1, max_iter=100, multi_class='ovr',
           penalty='l2', random_state=0, solver='liblinear',
tol=0.0001,
```

```
            verbose=0))]),
 'pipeline-1__clf': LogisticRegression(C=0.001, class_weight=None,
dual=False, fit_intercept=True,
            intercept_scaling=1, max_iter=100, multi_class='ovr',
            penalty='l2', random_state=0, solver='liblinear',
tol=0.0001,
            verbose=0),
 'pipeline-1__clf__C': 0.001,
 'pipeline-1__clf__class_weight': None,
 'pipeline-1__clf__dual': False,
 [...]
 'pipeline-1__sc__with_std': True,
 'pipeline-2': Pipeline(steps=[('sc', StandardScaler(copy=True, with_
mean=True, with_std=True)), ('clf', KNeighborsClassifier(algorithm='au
to', leaf_size=30, metric='minkowski',
             metric_params=None, n_neighbors=1, p=2,
weights='uniform'))]),
 'pipeline-2__clf': KNeighborsClassifier(algorithm='auto', leaf_
size=30, metric='minkowski',
             metric_params=None, n_neighbors=1, p=2,
weights='uniform'),
 'pipeline-2__clf__algorithm': 'auto',
 [...]
 'pipeline-2__sc__with_std': True}
```

得到get_params方法的返回值后，我们现在知道怎样去访问成员分类器的属性了。出于演示的目的，先通过网格搜索来调整逻辑斯谛回归分类器的正则化系数C以及决策树的深度。代码如下：

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {'decisiontreeclassifier__max_depth': [1, 2],
...           'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                     param_grid=params,
...                     cv=10,
...                     scoring='roc_auc')
>>> grid.fit(X_train, y_train)
```

完成网格搜索后，我们可以输出不同超参值的组合，以及10折交叉验证计算得出的平均ROC AUC得分。代码如下：

```
>>> for params, mean_score, scores in grid.grid_scores_:
...     print("%0.3f+/-%0.2f %r"
...             % (mean_score, scores.std() / 2, params))
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__
max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_
depth': 1}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__
max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__
max_depth': 2}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_
depth': 2}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__
max_depth': 2}
```

```
>>> print('Best parameters: %s' % grid.best_params_)
Best parameters: {'pipeline-1__clf__C': 100.0,
'decisiontreeclassifier__max_depth': 1}
```

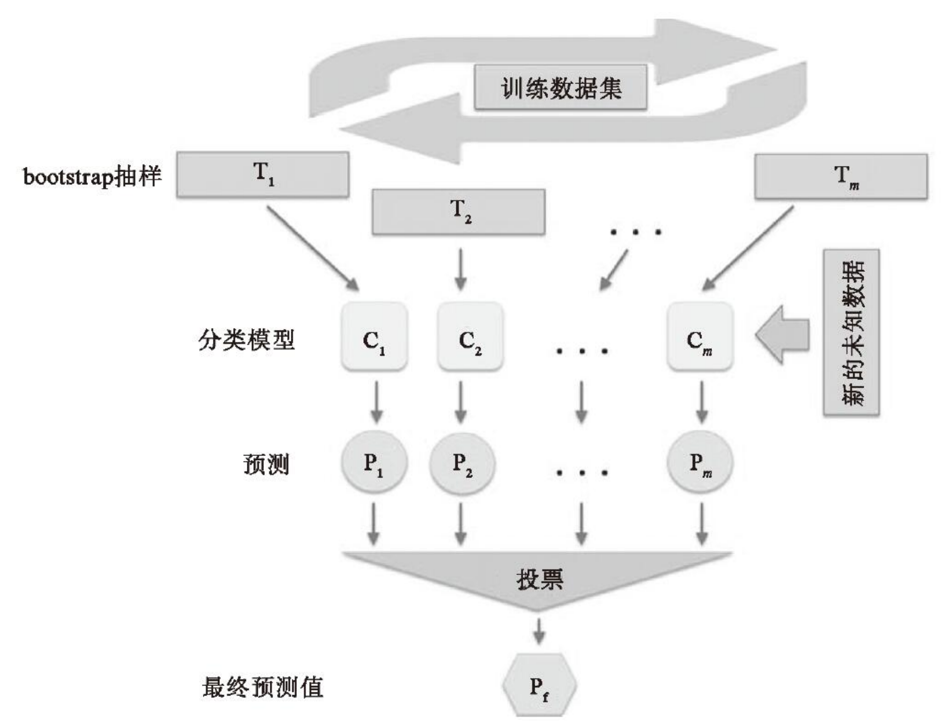
```
>>> print('Accuracy: %.2f' % grid.best_score_)
Accuracy: 1.00
```

正如我们所见，当选择的正则化强度较小时（C = 100.0），我们能够得到最佳的交叉验证结果，而决策树的深度似乎对性能没有任何影响，这意味着使用单层决策树足以对数据进行划分。请注意，在模型评估时，不止一次使用测试集并非一个好的做法，本节不打算讨论超参调优后的集成分类器泛化能力的评估。我们将继续学习另外一种集成方法：bagging。

我们在本节实现的多数投票方法有时也称为堆叠（stacking）。不过，堆叠算法更典型地应用于组合逻辑斯谛回归模型，以各独立分类器的输出作为输入，通过对这些输入结果的继承来预测最终的类标，详情请参阅David H.Wolpert的论文：Stacked generalization.Neural networks,5（2）:241——259，1992。

7.4 bagging——通过bootstrap样本构建集成分类器

bagging是一种与上一节实现的MajorityVoteClassifier关系紧密的集成学习技术，如下图所示：



不过，此算法没有使用相同的训练集拟合集成分类器中的单个成员分类器。由于原始训练集使用了boostrap抽样（有放回的随机抽样），这也就是bagging被称为boostrap aggregating的原因。为了用更具体的例子来解释boostrapping的工作原理，我们使用下图所示的示例来说明。在此，我们有7个不同的训练样例（使用索引1～7来表示），在每一轮的bagging循环中，它们都被可放回随机抽样。每个bootstrap抽样都被用于分类器C_j

的训练，这就是一棵典型的未剪枝的决策树：

样本索引	第1轮bagging	第2轮bagging	...
1	2	7	…
2	2	3	…
3	1	2	…
4	3	1	…
5	7	1	…
6	2	7	…
7	4	7	…
	C_1	C_2	C_m

bagging还与我们在第3章中曾经介绍过的随机森林紧密相关。实际上，随机森林是bagging的一个特例，它使用了随机的特征子集去拟合单棵决策。bagging最早由Leo Breiman在1994年的一份技术报告中提出。他还表示，bagging可以提高不稳定模型的准确率，并且可以降低过拟合的程度。我强烈建议读者阅读L.Breiman的论文[1]，以更深入了解bagging。此文献可在网上免费获取。

为了检验一下bagging的实际效果，我们使用第4章中用到的葡萄酒数据集构建一个更复杂的分类问题。在此我们只考虑葡萄酒中的类别2和类

别3，且只选择Alcohol和Hue这两个特征。

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/wine/wine.data', header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash',
...                    'Magnesium', 'Total phenols',
...                    'Flavanoids', 'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>> df_wine = df_wine[df_wine['Class label'] != 1]
>>> y = df_wine['Class label'].values
>>> X = df_wine[['Alcohol', 'Hue']].values
```

接下来，我们将类标编码为二进制形式，并将数据集按照六四分的比例划分为训练集和测试集：

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.cross_validation import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test =\
...          train_test_split(X, y,
...                           test_size=0.40,
...                           random_state=1)
```

scikit-learn中已经实现了BaggingClassifier相关算法，我们可从ensemble子模块中导入使用。在此，我们将使用未经剪枝的决策树作为成员分类器，并在训练数据集上通过不同的boostrap抽样拟合500棵决策树：

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=None)
>>> bag = BaggingClassifier(base_estimator=tree,
...                         n_estimators=500,
...                         max_samples=1.0,
...                         max_features=1.0,
...                         bootstrap=True,
...                         bootstrap_features=False,
...                         n_jobs=1,
...                         random_state=1)
```

然后我们将计算训练数据集和测试数据集上的预测准确率，以比较bagging分类器与单棵未剪枝决策树的性能差异：

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 1.000/0.854
```

基于上述代码执行的结果可见，未经剪枝的决策树准确地预测了训练数据的所有类标；但是，测试数据上极低的准确率表明该模型方差过高（过拟合）：

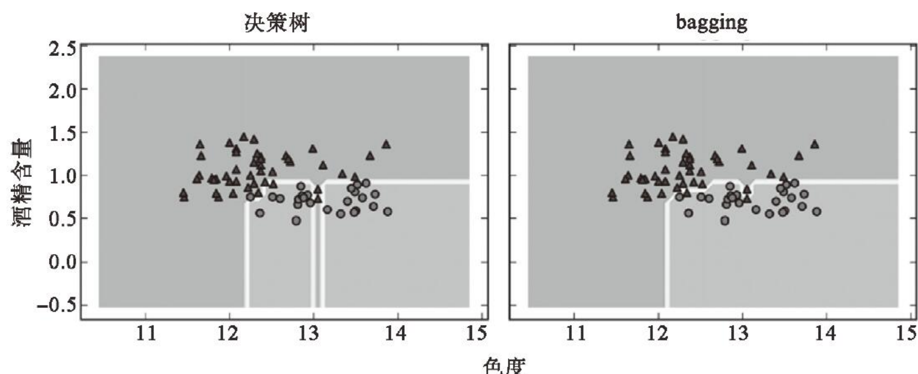
```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print('Bagging train/test accuracies %.3f/%.3f'
...       % (bag_train, bag_test))
Bagging train/test accuracies 1.000/0.896
```

虽然决策树与bagging分类器在训练集上的准确率相似（均为1.0），但是bagging分类器在测试数据上的泛化性能稍有胜出。下面我们比较一下决策树与bagging分类器的决策区域：

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, bag],


...                         ['Decision Tree', 'Bagging']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red', marker='o')
...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel(Alcohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...          s=Hue',
...          ha='center', va='center', fontsize=12)
>>> plt.show()
```

由结果图像可见，与深度为3的决策树线性分段边界相比，bagging集成分类器的决策边界显得更平滑：



本节我们只是简单通过示例了解了bagging。在实战中，分类任务会更加复杂，数据集维度会更高，使用单棵决策树很容易产生过拟合，这时bagging算法就可显示出其优势了。最后，我们需注意bagging算法是降低模型方差的一种有效方法。然而，bagging在降低模型偏差方面的作用不大，这也是我们选择未剪枝决策树等低偏差分类器作为集成算法成员分类器的原因。

[1] L.Breoman.Bagging Predictors.Machine Learing，24(2):123-140,1996.

7.5 通过自适应boosting提高弱学习机的性能

在本节对集成方法的介绍中，我们将重点讨论boosting算法中一个常用例子：AdaBoost（Adaptive Boosting的简称）。

AdaBoost背后的最初想法是由Robert Schapire于1990年提出的（R.E.Schapire.The Strength of Weak Learnability.Machine learning，5（2）:197——227，1990）。当Robert Schapire与Yoav Freund在第13届国际会议（ICML 1996）上发表AdaBoost算法后，随后的几年中，此算法就成为最广为应用的集成方法（Y.Freund,R.E.Schapire,et al.Experiments with a New Boosting Algorithm.In ICML,volume 96,pages 148——156,1996）。基于他们的开创性工作，Freund和Schapire在2003年获得了Goedel奖，这是发表计算机科学领域论文最负盛名的奖项。

在boosting中，集成分类器包含多个非常简单的成员分类器，这些成员分类器性能仅稍好于随机猜测，常被称作弱学习机。典型的弱学习机例子就是单层决策树。boosting主要针对难以区分的训练样本，也就是说，弱学习机通过在错误分类样本上的学习来提高集成分类的性能。与bagging不同，在boosting的初始化阶段，算法使用无放回抽样从训练样本中随机抽取一个子集。原始的boosting过程可总结为如下四个步骤：

1）从训练集D中以无放回抽样方式随机抽取一个训练子集d_1，用于弱学习机C_1的训练。

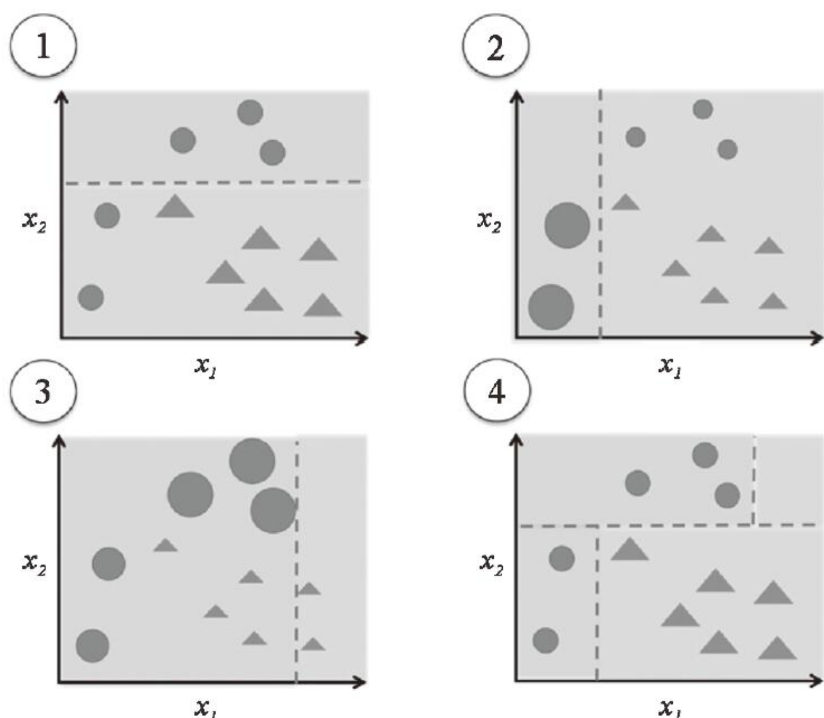
2）从训练集中以无放回抽样方式随机抽取第2个训练子集d_2，并将C_1中误分类样本的50%加入到训练集中，训练得到弱学习机C_2。

3）从训练集D中抽取C_1和C_2分类结果不一致的样本生成训练样本集d_3，以此训练第3个弱学习机C_3。

4）通过多数投票组合三个弱学习机C_1、C_2和C_3。

正如Leo Breiman所述[1]，与bagging模型相比，boosting可同时降低偏差和方差。在实践中，boosting算法（如AdaBoost）也存在明显的高方差问题，也就是说，对训练数据有过拟合倾向[2]。

AdaBoost与这里讨论的原始boosting过程不同，它使用整个训练集来训练弱学习机，其中训练样本在每次迭代中都会重新被赋予一个权重，在上一弱学习机错误的基础上进行学习进而构建一个更加强大的分类器。在深入到AdaBoost算法具体细节之前，先通过下图更深入了解一下AdaBoost的基本概念：



为循序渐进地介绍AdaBoost，我们从子图1开始，它代表了一个二类别分类问题的训练集，其中所有的样本都被赋予相同的权重。基于此训练集，我们得到一棵单层决策树（以虚线表示），它尝试尽可能通过最小化代价函数（或者是特殊情况下决策树集成中的不纯度得分）划分两类样本（三角形和圆形）。在下一轮（子图2）中，我们为前面误分类的样本（圆形）赋予更高的权重。此外，我们还降低被正确分类样本的权重。下一棵单层决策树将更加专注于具有最大权重的训练样本，也就是那些难于区分的样本。如子图2所示，弱学习机错误划分了圆形类的三个样本，它们在子图3中被赋予更大的权重。假定AdaBoost集成只包含3轮boosting过程，我们就能够用加权多数投票方式将不同重采样训练子集上得到的三个弱学习机进行组合，如子图4所示。

现在我们对AdaBoost的基本概念有了更好的认识，下面通过伪代码更深入地学习该算法。为了便于阐述，我们用十字符号（×）来表示向量的元素相乘，用点号（·）表示两个向量的内积，算法步骤如下：

1）以等值方式为权重向量 $\boldsymbol{w}$ 赋值，其中 $\sum_i w_i = 1$。

2）在 m 轮 boosting 操作中，对第 j 轮做如下操作：

3）训练一个加权的弱学习机：$C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$。

4）预测样本类标 $\hat{y} = \text{predict}(C_j, \boldsymbol{X})$。

5）计算权重错误率 $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} == \boldsymbol{y})$。

6）计算相关系数：$\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$。

7）更新权重：$\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$。

8）归一化权重，使其和为 1：$\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$。

9）完成最终的预测：$\hat{\boldsymbol{y}} = \left(\sum_{j=1}^{m} (\boldsymbol{\alpha}_j \times \text{predict}(C_j, \boldsymbol{X})) > 0 \right)$。

请注意，第 5 步中表达式 $(\hat{\boldsymbol{y}} == \boldsymbol{y})$ 的值为 1 或 0，其中 1 表示对应的预测结果正确，0 表示错误。

虽然AdaBoost算法看上去很简单，我们接下来用下面表格中的10个样本作为训练集，通过一个具体的例子来更深入地了解此算法：

样本索引	x	y	权重值	$\hat{y}$(x<=3.0)?	正确?	更新后的权重值
1	1.0	1	0.1	1	是	0.072
2	2.0	1	0.1	1	是	0.072
3	3.0	1	0.1	1	是	0.072
4	4.0	–1	0.1	–1	是	0.072
5	5.0	–1	0.1	–1	是	0.072
6	6.0	–1	0.1	–1	是	0.072
7	7.0	1	0.1	–1	是	0.167
8	8.0	1	0.1	–1	是	0.167
9	9.0	1	0.1	–1	是	0.167
10	10.0	–1	0.1	–1	是	0.167

表格的第1列为样本的索引序号，为1～10。假设此表格代表了一个一

维的数据集，第2列就相当于单个样本的值。第3列是对应于训练样本x_i的真实类标y_i，其中$y_i \in \{1,-1\}$。第4列为样本的初始权重，权重值相等，且通过归一化使其和为1。在训练样本数量为10的情况下，我们将权重向量w中的权值w_i都设定为0.1。假定我们的划分标准为$x \le 0.3$，第5列存储了预测类标$\hat{y}$。基于前面伪码给出的权重更新规则，最后一列存储了更新后的权重值。

乍一看，权重更新的计算似乎有些复杂，我们来逐步讲解计算过程。首先计算第5步中权重的错误率：

$$\varepsilon = 0.1\times0+0.1\times0+0.1\times0+0.1\times0+0.1\times0+0.1\times0+0.1\times0+0.1\times0+0.1\times0+0.1\times0 \\ = \frac{3}{10} = 0.3$$

下面来计算相关系数α_j（第6步），该系数将在第7步权重更新及最后一个步骤多数投票预测中作为权重来使用。

$$\alpha_j = \frac{0.5\log(1-\varepsilon)}{\varepsilon} \approx 0.424$$

在计算得到相关系数α_j后，我们可根据下述公式更新权重向量：

$$w := w \times \exp(-\alpha_j \times \hat{y} \times y)$$

其中，$\hat{y} \times y$ 为预测类标向量和真实类标向量逐元素相乘。由此，如果某个预测值 $\hat{y}_i$ 正确，则 $\hat{y}_i \times y_i$ 的符号为正，而且 α_i 本身值为正，则第 i 个权重会被降低：

$$0.1 \times \exp(-0.424 \times 1 \times 1) \approx 0.066$$

类似地，如果 $\hat{y}_i$ 预测的类标错误，则第 i 个权重将照如下方式更新：

$$0.1 \times \exp(-0.424 \times 1 \times (-1)) \approx 0.153$$

或者像这样：

$$0.1 \times \exp(-0.424 \times (-1) \times (1)) \approx 0.153$$

在完成所有权重向量值的更新后，我们通过归一化使所有权重的和为 1（第 8 个步骤）：

$$w := \frac{w}{\sum_i w_i}$$

其中，$\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$。

由此，对于分类正确的样本，其对应的权重在下一轮boosting中将从初始的0.1降为0.066/0.914≈0.072。同样，对于分类错误的样本，其对应权重将从0.1提高到0.153/0.914≈0.167。

简而言之，这就是AdaBoost。下面进入实践操作，通过scikit-learn训练一个AdaBoost集成分类器。我们仍将使用上一节中训练bagging元分类器的葡萄酒数据集。通过base_estimator属性，我们在500棵单层决策树上训练AdaBoostClassifier。

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               max_depth=1)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          learning_rate=0.1,
...                          random_state=0)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 0.845/0.854
```

由结果可见，与上一节中未剪枝决策树相比，单层决策树对于训练数据过拟合的程度更加严重一点：

```
>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('AdaBoost train/test accuracies %.3f/%.3f'
...       % (ada_train, ada_test))
AdaBoost train/test accuracies 1.000/0.875
```

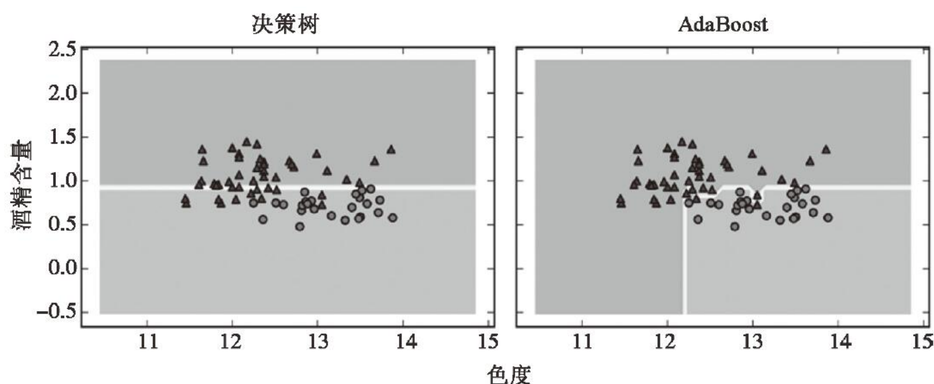
还可以看到，AdaBoost模型准确预测了所有的训练集类标，与单层决策树相比，它在测试集上的表现稍好。不过，在代码中也可看到，我们在降低模型偏差的同时使得方差有了额外的增加。

出于演示的目的，我们使用了另外一个简单的例子，但可以发现，与单层决策树相比，AdaBoost分类器能够些许提高分类性能，并且与上一节中训练的bagging分类器的准确率接近。不过请注意：重复使用测试集进行模型选择不是一个好的方法。集成分类器的泛化性能可能会被过度优化，对此我们在第6章中已经做过详细介绍。

最后，我们来看一下决策区域的形状：

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, ada],
...                         ['Decision Tree', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='red',
...                        marker='o')
... axarr[idx].set_title(tt)
... axarr[0].set_ylabel('Alcohol', fontsize=12)
>>> plt.text(10.2, -1.2,
...          s=Hue',
...          ha='center',
...          va='center',
...          fontsize=12)
>>> plt.show()
```

通过观察决策区域，我们可以看到Adabost的决策区域比单层决策树的决策区域复杂得多。此外，还注意到AdaBoost对特征空间的划分与上一节中训练的bagging分类器十分类似。



对集成技术做一个总结：毋庸置疑，与单独分类器相比，集成学习提高了计算复杂度。但在实践中，我们需仔细权衡是否愿意为适度提高预测性能而付出更多的计算成本。

关于预测性能与计算成本两者之间的权衡问题，一个常被提及的例子就是著名的一百万美元的奈飞竞赛（$1 Million Netflix Prize），最终胜出者就使用了集成技术。此算法的详细内容请参见A.Toescher等人的论文[3]。虽然获胜队伍得到了一百万美元的奖励，但由于模型的高复杂性，奈飞公司一直未将该模型投诸实际应用中[4]：

“……我们测得的准确率，额外的提高并不足以说服我们在工程方面进行投入，以将其应用到生产环境。”

[1] L. Breiman. Bias, Variance, and Arcing Classifiers. 1996.

[2] G. Raetsch, T. Onoda, and K. R. Mueller. An Improvement of Adaboost to Avoid Overfitting. In Proc. of the Int. Conf. on Neural Information Processing. Citeseer, 1998.

[3] A. Toescher, M. Jahrer, and R. M. Bell. The Bigchaos Solution to the

Netflix Grand Prize. Netflix prize documentation, 2009（读者也可通过http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf 获取。

[4] 请见http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html。

本章小结

在本章中，我们介绍了集成学习领域中最热门且应用最广泛的技术。集成方法通过组合不同的分类器模型来抵消各分类器固有的缺陷，通常能够得到一个稳定且性能优异的模型，因此在业界和机器学习竞赛领域得到了广泛的追捧。

在本章的开始，我们用Python实现了一个MajorityVoteClassifier类，它可以通过组合不同的算法得到一个分类器。进而我们学习了bagging，它能够在训练集上通过bootstrap进行随机抽样，并以多数投票为准则组合多个单独训练的成员分类器，成为一种能够有效降低模型方差的模型。然后我们讨论了AdaBoost，它是一种基于弱学习机的算法，能够从前一个弱学习机错误中进行学习。

在前面的章节中，我们讨论了各种学习算法、调优和评估技术。后面的章节，我们将着眼于机器学习的一个特定应用——情感分析，这已成为互联网和社交媒体时代一个有趣的话题。

第8章　使用机器学习进行情感分析

互联网和社交媒体在我们的生活中无处不在，人们的观点、评论及建议已经成为政治和商业领域的宝贵资源。得益于现代技术，我们能够高效地收集和分析此类数据。在本章中，我们将深入研究自然语言处理（natural language processing，NLP）领域的一个分支——情感分析（sentiment analysis），还将学习如何使用机器学习算法基于文档的情感倾向（polarity）——作者的观点——对文档进行分类。本章将涉及如下主题：

·清洗和准备文本数据

·基于文本文档构建特征向量

·训练机器学习模型用于区分电影的正面与负面评论

·使用out-of-core学习处理大规模文本数据集

8.1　获取IMDb电影评论数据集

情感分析，有时也称为观点挖掘（opinion mining），是NLP领域一个非常流行的分支；它分析的是文档的情感倾向（polarity）。情感分析的一个常见任务就是根据作者对某一主题所表达的观点或是情感来对文档进行分类。

在本章中，我们将使用由Maas等人[1]收集的互联网电影数据库（Internet Movie Database，IMDb）中的大量电影评论数据。此数据集包含50000个关于电影的正面或负面的评论，正面的意思是影片在IMDb数据库中的评分高于6星，而负面的意思是影片的评分低于5星。在本章后续内容中，我们将学习如何从这些电影评论的子集中抽取有意义的信息，以此来构建模型并用于预测评论者对影片的喜好。

读者可通过链接http://ai.stanford.edu/～amaas/data/sentiment/下载电影评论数据集的gzip压缩文档（84.1MB）：

·如果读者使用的是Linux或者macOS，可以打开一个终端窗口，使用cd命令定位到download目录，并执行tar-zxf aclImdb_v1.tar.gz命令解压数据集。

·若使用Windows，可以下载免费软件（如7-Zip[2]）解压下载的压缩文件。

在成功提取数据集后，我们现在着手将从压缩文件中得到的各文本文档组合为一个CSV文件。在下面的代码中，我们把电影的评论读取到

pandas的DataFrame对象中，使用普通的计算机该过程大约需要10分钟的时间。为了实现对处理过程的可视化，同时能够预测剩余处理时间，这里会用到PyPrind包（Python Progress Indicator,https://pypi.python.org/pypi/PyPrind/）。读者可以执行pip install pyprind命令安装PyPrind。

```
>>> import pyprind
>>> import pandas as pd
>>> import os
>>> pbar = pyprind.ProgBar(50000)
>>> labels = {'pos':1, 'neg':0}
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
...         path ='./aclImdb/%s/%s' % (s, l)
...         for file in os.listdir(path):
...             with open(os.path.join(path, file), 'r') as infile:
...                 txt = infile.read()
...            df = df.append([[txt, labels[l]]], ignore_index=True)
...             pbar.update()
>>> df.columns = ['review', 'sentiment']
0%                          100%
[##############################] | ETA[sec]: 0.000
Total time elapsed: 725.001 sec
```

执行上述代码，我们首先初始化了一个包含50000次迭代的进度条对象pbar，这也是我们准备读取的文档的数量。使用嵌套的for循环，我们迭代地读取aclImdb目录下的train和test两个子目录，以及pos和neg二级子目录下的文本文件，并将其附加到DataFrame对象df中，同时加入的还有文档对应的整数型类标（1代表正面，0代表负面）。

由于集成处理过后数据集中的类标是经过排序的，我们现在将使用np.random子模块下的permutation函数对DataFrame对象进行重排——由于在后续小节中我们将直接从本地驱动器读取数据，因此重排对于后续将数据集划分为训练集和测试集的操作是非常有用的。出于使用方便的考虑，我们将经过集成处理及重排后的评论数据集均存储为CSV文件：

```
>>> import numpy as np
>>> np.random.seed(0)
>>> df = df.reindex(np.random.permutation(df.index))
>>> df.to_csv('./movie_data.csv', index=False)
```

本章后续内容中会使用该数据集，现在读取并输出前三个样本的摘要，以此来快速确认数据已按正确格式存储：

```
>>> df = pd.read_csv('./movie_data.csv')
>>> df.head(3)
```

如果在IPython Notebook上运行上述代码，那么将看到数据集中前三个样本的信息，如下表所示：

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

[1] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. Learning Word Vectors for Sentiment Analysis. In the proceedings of the 49th Annual Meeting of the Association for Computational Linguistics:Human Language Technologies, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[2] http://www.7-zip.org.

8.2 词袋模型简介

记得在第4章中，我们需要将文本或者单词等分类数据转换为数值格式，以方便在机器学习算法中使用。在本节，我们将介绍词袋模型（bag-of-words model），它将文本以数值特征向量的形式来表示。词袋模型的理念很简单，可描述如下：

1）我们在整个文档集上为每个词汇创建了唯一的标记，例如单词。

2）我们为每个文档构建一个特征向量，其中包含每个单词在此文档中出现的次数。

由于每个文档中出现的单词数量只是整个词袋中单词总量很小的一个子集，因此特征向量中的大多数元素为零，这也是我们称之为稀疏（sparse）的原因。这些内容听起来过于抽象，不过毋庸担心，下面我们将逐步讲解创建简单词袋模型的过程。

8.2.1 将单词转换为特征向量

如需根据每个文档中的单词数量构建词袋模型，我们可以使用scikit-learn中的Count-Vectorizer类。如下述代码所示，CountVectorizer以文本数据数组作为输入，其中文本数据可以是个文档或仅仅是个句子，返回的就是我们所要构建的词袋模型：

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array([
...         'The sun is shining',
...         'The weather is sweet',
...         'The sun is shining and the weather is sweet'])
>>> bag = count.fit_transform(docs)
```

通过调用CountVectorizer的fit_tranform方法，我们创建了词袋模型的词汇库，并将下面三个句子转换成为稀疏的特征向量：

1. `The sun is shining`
2. `The weather is sweet`
3. `The sun is shining and the weather is sweet`

我们将相关词汇的内容显示出来，以更好地理解相关概念：

```
>>> print(count.vocabulary_)
{'the': 5, 'shining': 2, 'weather': 6, 'sun': 3, 'is': 1, 'sweet': 4,
'and': 0}
```

由上述命令的运行结果可见，词汇以Python字典的格式存储，将单个的单词映射为一个整数索引。下面我们来看一下之前创建的特征向量：

```
>>> print(bag.toarray())
[[0 1 1 1 0 1 0]
 [0 1 0 0 1 1 1]
 [1 2 1 1 1 2 1]]
```

特征向量中的每个索引位置与通过CountVectorizer得到的词汇表字典中存储的整数值对应。例如，索引位置为0的第一个特征对应单词and，它只在最后一个文档中出现过；单词is的索引位置是1（文档向量中的第二个特征），它在三个句子中都出现过。出现在特征向量中的值也称作原始词频（raw term frequency）：tf（t,d）——词汇t在文档d中出现的次数。

我们刚创建的词袋模型中，各项目的序列也称为1元组（1-gram）或者单元组（unigram）模型——词汇表中的每一项或者每个单元代表一个词汇。更普遍地，自然语言处理（NLP）中各项（包括单词、字母、符号）的序列，也称作n元组（n-gram）。n元组模型中数字n的选择依赖于特定的应用；例如，Kanaris等人通过研究发现，在反垃圾邮件过滤中，n的值为3或者4的n元组即可得到很好的效果（Ioannis Kanaris,Konstantinos Kanaris,Ioannis Houvardas,and Efstathios Stamatatos.Words vs Character N-Grams for Anti-Spam Filtering.International Journal on Artificial Intelligence Tools,16（06）:1047–1067,2007）。对n元组表示的概念做个总结，分别使用1元组和2元组来表示文档"the sum is shining"的结果如下：

·1元组："the"，"sun"，"is"，"shining"

·2元组："the sun"，"sun is"，"is shining"

借助于scikit-learn中的CountVecorizer类，我们可以通过设置其

ngram_range参数来使用不同的n元组模型。此类默认为1元组，使用ngram_range=（2,2）初始化一个新的CountVectorizer类，可以得到一个z元组表示。

8.2.2 通过词频-逆文档频率计算单词关联度

当我们分析文本数据时，经常遇到的问题就是：一个单词出现在两种类型的多个文档中。这种频繁出现的单词通常不包含有用或具备辨识度的信息。在本小节中，我们将学习一种称为词频-逆文档频率（term frequency-inverse document frequency,tf-idf）的技术，它可以用于解决特征向量中单词频繁出现的问题。tf-idf可以定义为词频与逆文档频率的乘积：

$$\text{tf-idf}(t,d) = \text{tf}(t,d) \times \text{idf}(t,d)$$

其中，tf（t,d）是我们上一节中介绍的词频，而逆文档频率idf（t,d）可通过如下公式计算：

$$\text{idf}(t, d) = \log \frac{n_d}{1 + \text{df}(d, t)}$$

这里的n_d为文档的总数，df（d,t）为包含词汇t的文档d的数量。请注意，分母中加入常数1是可选的，对于没有出现在任何训练样本中的词汇，它能保证分母不为零；取对数是为了保证文档中出现频率较低的词汇不会被赋予过大的权重。

scikit-learn还实现了另外一个转换器：TfidfTransformer，它以CountVectorizer的原始词频作为输入，并将其转换为tf-idf：

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tfidf = TfidfTransformer()
>>> np.set_printoptions(precision=2)
>>> print(tfidf.fit_transform(count.fit_transform(docs)).toarray())
[[ 0.    0.43  0.56  0.56  0.    0.43  0.  ]
 [ 0.    0.43  0.    0.    0.56  0.43  0.56]
 [ 0.4   0.48  0.31  0.31  0.31  0.48  0.31]]
```

在上一小节中我们看到，is在第三个文档中具有最高的词频，它是文档中出现得最为频繁的单词。但是在将特征向量转换为tf-idf后，单词is在第三个文档中只得到了一个相对较小的tf-idf（0.48），这是由于第一和第二个文档中都包含单词is，因此它不太可能包含有用或是有辨识度的信息。

不过，如果我们人工计算特征向量中单个条目的tf-idf值，就会发现，TfidfTrans-former中对tf-idf的计算方式与我们此前定义的标准计算公式不同。scikit-learn中实现的idf和tf-idf分别为：

$$\text{idf}(t, d) = \log \frac{1+n_d}{1+\text{df}(d, t)}$$

在scikit-learn中使用的tf-idf公式为：

tf-idf（t,d）= tf（t,d）×（idf（t,d）+1）

通常在计算tf-idf之前都会对原始词频进行归一化处理，TfidfTransformer就直接对tf-idf做了归一化。默认情况下（norm='l2'），scikit-learn中的TfidfTransformer使用L2归一化，它通过与一个未归一化特征向量L2范数的比值，使得返回向量的长度为1：

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2+v_2^2+\cdots+v_n^2}} = \frac{v}{\left(\sum_{i=1}^{n} v_i\right)^{1/2}}$$

为确保读者能够理解TfidfTransformer的工作方式，在此我们给出一个例子，计算第三个文档中单词is的tf-idf。

在文档3中，单词is的词频为2（tf = 2），由于is在三个文档中都出现过，因此它的文档频率为3（df = 3）。由此，idf的计算方法如下：

$$\text{idf}("\text{is}", d3) = \log\frac{1+3}{1+3} = 0$$

为了计算tf-idf，我们需要为逆文档频率的值加1，并且将其与词频相乘：

tf-idf（"is",d3） = 2×（0 + 1） = 2

如果我们对第三个文档中的所有条目重复此过程，将会得到如下tf-idf向量：[1.69,2.00,1.29,1.29,1.29,2.00,1.29]。大家应该注意到了，这里得到的特征向量的值不同于此前我们使用TfidfTransformer得到的值。最后还缺少一个对tf-idf进行L2归一化的步骤，计算方式如下：

$$\text{tf-idf}("\text{is}", d3)_{norm} \frac{[1.69, 2.00, 1.29, 1.29, 1.29, 2.00, 1.29]}{\sqrt{1.69^2, 2.00^2+1.29^2+1.29^2+1.29^2+2.00^2+1.29^2}}$$

$$=[0.40, 0.48, 0.31, 0.31, 0.31, 0.48, 0.31]$$

可以看到，现在的结果与使用scikit-learn中TfidfTransformer得到的值相同。既然已经理解了tf-idf的计算方式，我们进入下一节，将这些概念应用于电影评论数据集。

8.2.3 清洗文本数据

在前面的小节中，我们学习了词袋模型、词频以及逆文档频率。不过在构建词袋模型之前，最重要的一步就是——通过去除所有不需要的字符对文本数据进行清洗。为了说明此步骤的重要性，我们先展示一下经过重排后数据集中第一个文档的最后50个字符：

```
>>> df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

正如我们所见，文本中包含HTML标记、标点符号以及其他非字母字符。HTML标记并未包含很多有用语义，标点符号可以在某些NLP语境中提供有用及附加信息。不过，为了简单起见，我们将去除标点符号，只保留表情符号，如":)"，因为它们在情感分析中通常是有用的。为了完成此任务，我们将使用Python的正则表达式（regex）库：re，代码如下：

```
>>> import re
>>> def preprocessor(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)
...     text = re.sub('[\W]+', ' ', text.lower()) + \
...             '.join(emoticons).replace('-', '')
...     return text
```

通过代码中的第一个正则表达式<[^>]*>，我们试图移除电影评论中所有的HTML标记。虽然许多程序员通常建议不要使用正则表达式解析HTML，但这个正则表达式足以清理此特定数据集。在移除了HTML标记后，我们使用稍微复杂的正则表达式寻找表情符号，并将其临时存储在emoticons中。接下来，我们通过正则表达式[\W]+删除文本中所有的非单词字符，将文本转换为小写字母，最后将emoticons中临时存储的表情

符号追加在经过处理的文档字符串后。此外，为了保证表情符号的一致，我们还删除了表情符号中代表鼻子的字符（-）。

虽然正则表达式为我们提供了一种在字符串中搜索特定字符的方便且有效的方法，但掌握它会有一个相对陡峭的学习曲线，而对正则表达式的讨论也超出了本书的范围。不过，读者可以通过谷歌开发者门户找到相关的入门教程：https://developers.google.com/edu/python/regular-expressions，或者查看Python中re模块的官方文档：https://docs.python.org/3.4/library/re.html。

尽管将表情符号追加到经过清洗的文档字符串的最后，看上去不是最简洁的方法，但是当词袋模型中所有的符号都由单个单词组成时，单词的顺序并不重要。在对文档划分成条目、单词及符号做进一步讨论之前，先来确认一下预处理操作是否能正常工作：

```
>>> preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
>>> preprocessor("</a>This :) is :( a test :-)!")
'this is a test :) :( :)'
```

最后，由于我们将在下一节中反复使用在此经过清洗的文本数据，现在通过preprocessor函数移除DataFrame中所有的电影评论信息：

```
>>> df['review'] = df['review'].apply(preprocessor)
```

8.2.4 标记文档

准备好电影评论数据集后，我们需要思考如何将文本语料拆分为单独的元素。标记（tokenize）文档的一种常用方法就是通过文档的空白字符将其拆分为单独的单词。

```
>>> def tokenizer(text):
...     return text.split()
>>> tokenizer('runners like running and thus they run')
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

在对文本进行标记的过程中，另外一种有用的技术就是词干提取（word stemming），这是一个提取单词原形的过程，那样我们就可以将一个单词映射到其对应的词干上。最初的词干提取算法是由Martin F.Porter于1979年提出的，由此也称为Porter Stemmer算法[1]。Python自然语言工具包（NLTK,http://www.nltk.org）实现了Porter Stemming算法，我们将在下一小节中用到它。要安装NLTK，只要执行命令：pip install nltk。

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
>>> def tokenizer_porter(text):
...     return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

尽管NLTK不是本章的重点，若读者对自然语言处理有更进一步的兴趣，作者强烈建议访问NTLK的网站及其官方资料，它们均可通过链接http://www.nltk.org/book/免费获得。

使用nltk包中的PorterStemmer修改tokenizer函数，使单词都恢复到其原始形式，借用前面的例子，单词running恢复为run。

Porter stamming算法可能是最原始也是最简单的词干提取算法了。其他流行的词干提取算法包括Snowball stemmer（Porter2，也称为"English"stemmer）以及Lancaster stemmer（Paice-Husk stemmer），与Porter stemming算法相比，它们提取速度更高不过提取时也更加野蛮。这些算法也在nltk包中得以实现（http://www.nltk.org/api/nltk.stem.html）。

正如前面例子所示，词干提取也可能生成一些不存在的单词，例如前面例子中提到的thu（通过提取thus得到），词形还原（lemmatization）是一种以获得单个单词标准形式（语法上正确）——也就是所谓的词元（lemma）——为目标的技术。不过，相较于词干提取，词形还原的计算更加复杂和昂贵，通过实际应用中的观察发现，在文本分类中，这两种技术对分类结果的影响不大（Michal Toman,Roman Tesar,and Karel Jezek.Influence of word normalization on text classification.Proceedings of InSciT,pages 354-358,2006）。

下一章将使用词袋模型训练一个机器学习模型，在此之前，我们简要介绍下另一种有用的技术：停用词移除（stop-word removal）。停用词是指在各种文本中太过常见，以致没有（或很少）含有用于区分文本所属类别的有用信息。常见的停用词有is、and、has等。由于tf-idf可以降低频繁出现单词的权重，因此当我们使用原始或归一化的词频而不是tf-idf时，移除停用词是很有用的。

我们可通过调用nltk.download函数得到NLTK库提供的停用词，并使用其中的127个停用词对电影评论数据进行停用词移除处理：

```
>>> import nltk
>>> nltk.download('stopwords')
```

完成之后，可以通过如下方式加载和使用由nltk.download得到的英文停用词集：

```
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>>  [w for w in tokenizer_porter('a runner likes running and runs a
lot')[-10:] if w not in stop]

['runner', 'like', 'run', 'run', 'lot']
```

[1] Martin F.Porter.An algorithm for suffix stripping.Program:electronic library and information systems,14(3):130-137.1980.

8.3 训练用于文档分类的逻辑斯谛回归模型

在本节中，我们将训练一个逻辑斯谛回归模型以将电影评论划分为正面评价或负面评价。首先，将上一节中清洗过的文本文档对象DataFrame划分为25000个训练文档和25000个测试文档：

```
>>> X_train = df.loc[:25000, 'review'].values
>>> y_train = df.loc[:25000, 'sentiment'].values
>>> X_test = df.loc[25000:, 'review'].values
>>> y_test = df.loc[25000:, 'sentiment'].values
```

接下来我们将使用GridSearchCV对象，并使用5折分层交叉验证找到逻辑斯谛回归模型最佳的参数组合：

```
>>> from sklearn.grid_search import GridSearchCV
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> tfidf = TfidfVectorizer(strip_accents=None,
...                         lowercase=False,
...                         preprocessor=None)
>>> param_grid = [{'vect__ngram_range': [(1,1)],
...               'vect__stop_words': [stop, None],
...               'vect__tokenizer': [tokenizer,
...                                   tokenizer_porter],
...               'clf__penalty': ['l1', 'l2'],
...               'clf__C': [1.0, 10.0, 100.0]},
...              {'vect__ngram_range': [(1,1)],
...               'vect__stop_words': [stop, None],
...               'vect__tokenizer': [tokenizer,
...                                   tokenizer_porter],
...               'vect__use_idf':[False],
...               'vect__norm':[None],
...               'clf__penalty': ['l1', 'l2'],
...               'clf__C': [1.0, 10.0, 100.0]}
...              ]
>>> lr_tfidf = Pipeline([('vect', tfidf),
...                      ('clf',
...                       LogisticRegression(random_state=0))])
>>> gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
...                            scoring='accuracy',
...                            cv=5, verbose=1,
...                            n_jobs=-1)
>>> gs_lr_tfidf.fit(X_train, y_train)
```

在上述代码中使用参数组合初始化GridSearchCV对象时，我们严格限制了参数组合的数量，这是因为涉及大量的特征向量，以及数量众多的单词，会导致网格搜索的计算成本很高，使用普通的计算机，完成网格搜索大约需要40分钟的时间。

在上述示例代码中，由于TfidfVectornizer组合使用了CountVectorizer和TfidfTrans-former，因此我们用TfidfVectorizer替代后面的两个类。param_grid包含两个参数字典。在第一个字典中，我们使用了TfidfVectorizer的默认设置（use_idf＝True,smooth_idf＝True，以及norm＝'l2'）计算tf-idf；我们在第二个字典中将参数设置为use_idf＝False,smooth_idf＝False，以及norm＝None，以在原始词频上完成模型的训练。此外，对于逻辑斯谛回归算法，我们通过罚项使用了L2和L1正则化，并通过为逆正则参数C定义一个取值区间来比较不同正则化强度之间的差异。

在网格搜索结束后，我们可以输出最佳的参数集：

```
>>> print('Best parameter set: %s ' % gs_lr_tfidf.best_params_)
Best parameter set: {'clf__C': 10.0, 'vect__stop_words': None,
'clf__penalty': 'l2', 'vect__tokenizer': <function tokenizer at
0x7f6c704948c8>, 'vect__ngram_range': (1, 1)}
```

在此可见，网格搜索返回的最佳参数设置集合为：使用不含有停用词的常规标记（token）生成器，同时在逻辑斯谛回归中使用tf-idf，其中逻辑斯谛回归分类器使用L2正则化，正则化强度C = 10.0。

使用网格搜索得到的最佳模型，我们分别输出训练集上5折交叉验证的准确率得分，以及在测试数据集上的分类准确率：

```
>>> print('CV Accuracy: %.3f'
...       % gs_lr_tfidf.best_score_)
CV Accuracy: 0.897
>>> clf = gs_lr_tfidf.best_estimator_
>>> print('Test Accuracy: %.3f'
...     % clf.score(X_test, y_test))
Test Accuracy: 0.899
```

结果表明，我们的机器学习模型针对电影评论是正面评价还是负面评价的分类准确率为90%。

朴素贝叶斯分类器（Na黠e Bayes classifier）是迄今为止执行文本分类十分流行的一种分类器，特别是用于垃圾邮件过滤。朴素贝叶斯分类器易于实现，计算性能高效，相对于其他算法，它在小数据集上的表现异常出色。虽然本书并未讨论朴素贝叶斯分类器，有兴趣的读者可以阅读我在arXiv上关于朴素贝叶斯分类的论文（S.Raschka.Naive Bayes and Text Classification I-introduction and Theory.Computing Research Repository（CoRR）,abs/1410.5329,2014.http://arxiv.org/pdf/1410.5329v3.pdf）。

8.4 使用大数据——在线算法与外存学习

如果你执行上一节中的示例代码，就会发现：在网格搜索阶段为包含50000个电影评论的数据集构建特征向量的计算成本很高。在许多真实应用中，经常会用到更大的数据集，数据集大小甚至会超出计算机内存的容量。并不是每个人都有机会使用超级计算设备，因此我们将通过一种称为外存学习的技术来处理超大数据集。

回顾一下第2章中我们曾经介绍过的随机梯度下降（stochastic gradient descent）的概念，此优化算法每次使用一个样本更新模型的权重信息。在本节，我们将使用scikit-learn中SGDClassifier的partial_fit函数来读取本地存储设备，并且使用小型子批次（minibatches）文档来训练一个逻辑斯谛回归模型。

在本章开始时，我们构建了movie_data.csv数据文件，且在移除停用词阶段对此单词做了token处理。首先，我们定义一个tokenizer函数来清理movie_data.csv文件中未经处理的文本数据：

```
>>> import numpy as np
>>> import re

>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> def tokenizer(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
...                            text.lower())
...     text = re.sub('[\W]+', ' ', text.lower()) \
...            + ' '.join(emoticons).replace('-', '')
...     tokenized = [w for w in text.split() if w not in stop]
...     return tokenized
```

接下来我们定义一个生成器函数：stream_docs，它每次读取且返回一个文档的内容：

```
>>> def stream_docs(path):
...     with open(path, 'r') as csv:
...         next(csv) # skip header
...         for line in csv:
...             text, label = line[:-3], int(line[-2])
...             yield text, label
```

为了验证stream_docs函数是否能正常工作，我们来读取一下movie_data.csv文件的第一个文档，它应返回一个包含评论信息和对应类标的元组：

```
>>> next(stream_docs(path='./movie_data.csv'))
('"In 1974, the teenager Martha Moxley ... ',1)
```

定义一个get_minibatch函数，它以stream_doc函数得到的文档数据流作为输入，并通过参数size返回指定数量的文档内容：

```
>>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...         try:
...             for _ in range(size):
...                 text, label = next(doc_stream)
...                 docs.append(text)
...                 y.append(label)
...         except StopIteration:
...             return None, None
...         return docs, y
```

不幸的是，由于需要将所有的词汇加载到内存中，我们无法通过CountVectorizer来使用外存学习方法。另外，TfidfVectorizer需要将所有训练数据集中的特征向量加载到内存以计算逆文档频率。不过，scikit-learn提供了另外一个处理文本信息的向量处理器：HashingVectorizer。HashingVectorizer是独立于数据的，其哈希算法使用了Austin Appleby提

出的32位MurmurHash3算法（https://sites.google.com/site/murmurhash/）。

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> from sklearn.linear_model import SGDClassifier
>>> vect = HashingVectorizer(decode_error='ignore',
...                          n_features=2**21,
...                          preprocessor=None,
...                          tokenizer=tokenizer)
>>> clf = SGDClassifier(loss='log', random_state=1, n_iter=1)
>>> doc_stream = stream_docs(path='./movie_data.csv')
```

在上述代码中，我们使用tokenizer函数初始化HashingVectorizer，并且将特征的数量设定为2^{21}。此外，我们通过将SGDClassifier中loss参数的值设定为log来重新初始化逻辑斯谛回归分类器。请注意：通过为HashingVectorizer设定一个大的特征数量，降低了哈希碰撞的概率，不过同时也增加了逻辑斯谛回归模型中系数的数量。

现在到了真正有趣的部分。设置好所有的辅助函数后，我们可以通过下述代码使用外存学习：

```
>>> import pyprind
>>> pbar = pyprind.ProgBar(45)
>>> classes = np.array([0, 1])
>>> for _ in range(45):
...     X_train, y_train = get_minibatch(doc_stream, size=1000)
...     if not X_train:
...         break
...     X_train = vect.transform(X_train)
...     clf.partial_fit(X_train, y_train, classes=classes)
...     pbar.update()
0%                          100%
[##############################] | ETA[sec]: 0.000
Total time elapsed: 50.063 sec
```

为了估计学习算法的进度，我们再一次使用PyPrind包。将进度条对象设定为45次迭代，在接下来的for循环中，我们在45个文档的子:批次上

进行迭代，每个子:批次包含1000个文档。

完成了增量学习后，我们将使用剩余的5000个文档来评估模型的性能：

```
>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
>>> X_test = vect.transform(X_test)
>>> print('Accuracy: %.3f' % clf.score(X_test, y_test))
Accuracy: 0.868
```

可以看到，模型的准确率约为87%，略微低于我们在上一节使用网格搜索进行超参调优得到的模型。不过外存学习的存储效率很高，只用了不到一分钟的时间就完成了计算。最后，我们可以通过剩下的5000个文档来升级模型：

```
>>> clf = clf.partial_fit(X_test, y_test)
```

如果读者希望直接进入第9章，建议不要关闭当前的Python会话窗口。在下一章中，我们学习如何将刚刚训练得到的模型存储到硬盘以供后续使用，以及如何将其嵌入到Web应用中。

虽然词袋模型仍旧是文本分类领域最为流行的模型，但是它没有考虑句子的结构和语法。一种流行的词袋模型扩展就是潜狄利克雷分配（Latent Dirichlet Allocation），这是一种考虑句子潜在语义的主题模型（D.M.Blei,A.Y.Ng,and M.I.Jordan.Latent Dirichlet allocation.The Journal of machine Learning research,3:993-1022,2003）。

word2vec是最近提出的一种词袋模型的替代算法，它由谷歌公司在2013年提出（T.Mikolov,K.Chen,G.Corrado,and J.Dean.Efficient Estimation of Word Represen-tations in Vector Space.arXiv preprint

arXiv:1301.3781,2013）。word2vec算法是基于神经网络的一种无监督算法，它会自动尝试学习单词之间的关系。word2vec背后的理念就是将词义相近的单词划分到相同的簇中；通过巧妙的向量间隔，此模型可以通过简单的向量计算来得到合适的单词，例如：king–man＋woman＝queen。

读者可以通过链接https://code.google.com/p/word2vec/找到word2vec模型原始的C语言实现、相关的论文及其替代实现代码。

本章小结

在本章中，我们学习了如何使用机器学习算法根据文本文档的情感倾向对其进行分类，这是自然语言处理领域中情感分析的基本工作。我们不仅学习了如何使用词袋模型对文档进行编码，而且学习了如何使用词频-逆文档频率来矫正词频权重。

在对文本进行情感分析的过程中，由于生成的特征向量巨大，导致文本数据处理会产生较高的计算成本。最后一节中，我们学习了外存和增量学习算法，它们无需将整个数据集同时加载到内存就能够完成对机器学习模型的训练。

在下一章，我们将使用文档分类器，并将学习如何将其嵌入到Web应用中。

第9章　在Web应用中嵌入机器学习模型

在前几章中，我们学习了多种机器学习的相关概念与算法，它们可以帮助我们更好、更高效地做出决策。然而，机器学习技术并不仅仅局限于离线应用和分析，它们也可以成为Web服务的预测引擎。例如，Web应用领域常用的机器学习模型包括：通过表单发送的垃圾邮件检测、搜索引擎，以及媒体、购物门户网站用到的推荐系统等。

在本章我们将学习如何将机器学习模型嵌入到Web应用中，不仅仅是分类，还包括从实时数据中学习。本章将涵盖如下主题：

·保存训练后的机器学习模型的状态

·使用SQLite数据库存储数据

·使用Flask框架开发Web应用程序

·在公共Web服务器上部署机器学习应用

9.1 序列化通过scikit-learn拟合的模型

正如我们在第8章中所讨论的那样，训练机器学习模型会带来很高的计算成本。当然，我们不希望每次进行预测分析都打开Python交互窗口来训练模型，或者重新加载Web应用程序。模型持久化的一个方法就是使用Python内嵌的pickle模块[1]，它使得我们可以在Python对象与字节码之间进行转换（序列化与反序列化），这样就可以将分类器当前的状态保存下来。当需要对新的样本进行分类时，可以直接加载已保存的分类器，而不必再次通过训练数据对模型进行训练。执行下列代码前，请确保已经使用第8章中介绍的方法完成了外存逻辑斯谛回归模型的训练，并且已在当前Python会话中：

```
>>> import pickle
>>> import os
>>> dest = os.path.join('movieclassifier', 'pkl_objects')
>>> if not os.path.exists(dest):
...     os.makedirs(dest)
>>> pickle.dump(stop,
...             open(os.path.join(dest, 'stopwords.pkl'),'wb'),
...             protocol=4)
>>> pickle.dump(clf,
...             open(os.path.join(dest, 'classifier.pkl'), 'wb'),

...             protocol=4)
```

通过上述代码，我们创建了一个movieclassifier目录，后续我们将在此存储Web应用中的文件和数据。在此目录下，我们创建了一个pkl_objects子目录，用于存储序列化后的Python对象。使用pickle中的dump方法，对训练好的逻辑斯谛回归模型及NLTK库中的停用词进行序列化，这样就不必在Web应用服务器上安装NLTK库了。dump方法的第一个

参数是我们要持久化的对象，第二个参数用于制定序列化对象的保存路径，在此我们使用了Python中的open方法。open方法中的wb参数代表以二进制方式存储数据，而protocol＝4则代表使用最新、最高效的protocol协议，此协议已在Python 3.4中使用[2]。

我们的逻辑斯谛回归模型中包含多个NumPy的数组，例如权重向量等，使用joblib库是序列化NumPy数组更加高效的一个替代方法。不过为了保障服务器环境与后续小节中的配置一致，我们仍旧使用标准的pickle方法。如果读者对joblib感兴趣，可以访问链接https://pypi.python.org/pypi/joblib以获得更多信息。

由于无需拟合HashingVectorizer，也就不必对其进行持久化操作。相反，我们可以创建一个新的Python脚本文件，通过此脚本可以将向量数据导入到当前Python会话中。请拷贝下列代码，并以vectorizer.py作为文件名，保存在movieclassifier目录下：

```
from sklearn.feature_extraction.text import HashingVectorizer
import re
import os
import pickle

cur_dir = os.path.dirname(__file__)
stop = pickle.load(open(
                os.path.join(cur_dir,
                'pkl_objects',
                'stopwords.pkl'), 'rb'))

def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
                           text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) \
                  + ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized

vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)
```

在对Python对象进行序列化操作以及保存好vectorizer.py文件后，最好重新启动一下Python交互窗口或者IPython Notebook，以测试是否可以准确无误地对已保存模型进行逆持久化操作。不过，请注意，由于pickle模块未对恶意代码进行防范，因此对从不受信任渠道获得的数据进行持久化操作可能会具有潜在的安全风险。在终端窗口中，定位到movieclassifier所在的路径，开启一个新的Python会话，并执行下列代码以确保可以正常导入vectorizer及对分类器进行逆持久化处理：

```
>>> import pickle
>>> import re
>>> import os
>>> from vectorizer import vect
>>> clf = pickle.load(open(
...         os.path.join('pkl_objects',
...                      'classifier.pkl'), 'rb'))
```

在成功加载vectorizer及反序列化分类器后，我们现在就可以使用这

些对象对文档样本进行预处理，并对它们所代表的情感倾向进行预测：

```
>>> import numpy as np
>>> label = {0:'negative', 1:'positive'}
>>> example = ['I love this movie']
>>> X = vect.transform(example)
>>> print('Prediction: %s\nProbability: %.2f%%' %\
...       (label[clf.predict(X)[0]],
...        np.max(clf.predict_proba(X))*100))
Prediction: positive
Probability: 91.56%
```

由于分类器返回的类标为整数，我们在此定义一个简单的Python字典将整数映射到对应的情感上。然后，我们使用HashingVectorizer将样本文档转换为单词向量X。最后，我们使用逻辑斯谛回归分类器的predict方法预测类标，并通过predict_proba方法返回各预测结果相应的概率。请注意，对predict_proba方法的调用会返回一个数组，以及每个类标所对应的概率。由于predict返回的是较高概率对应的类标，我们则使用np.max函数返回对应预测类别的概率[3]。

[1] https://docs.python.org/3.4/library/pickle.html.

[2] 如果无法正常使用protocol 4，请检查是否安装了Python 3的最新版本。此外，你也可以选择一个较低版本的protocol。

[3] 情绪的概率，最终的类别是通过np.max得到的概率较高的那个。——译者注

9.2　使用SQLite数据库存储数据

在本节中，我们将创建一个简单的SQLite数据库以收集Web应用的用户对于预测结果的反馈。基于这些反馈，我们可以对分类模型进行更新。SQLite是一款开源的SQL数据库引擎，由于它无需运行单独的服务器，因此成为小型项目和简单Web应用的理想选择。从本质上来说，SQLite数据库可以看作是一个单一的、自包含（不依赖于其他模块与组件）的数据库文件，它允许我们直接访问存储文件。此外，SQLite无需任何针对特定系统的设置，常用操作系统也都支持。其出色的可靠性为其赢得了良好的声誉，被广泛应用于Google、Mozilla、Adobe、Apple、Microsoft等知名公司。如果读者想详细了解SQLite，请访问其官方网站：http://www.sqlite.org。

幸运的是，由于Python的“自带电池”[1]哲学，在Python标准库中已经包含了支持SQLite的API：sqlite3，这使得我们可以直接操作使用SQLite数据库（关于sqlite3的更详细信息请参见https://docs.python.org/3.4/library/sqlite3.html）。

通过执行下列代码，我们将在movieclassifier所在目录创建一个新的SQLite数据库，并向其中插入两条电影评论的示例数据：

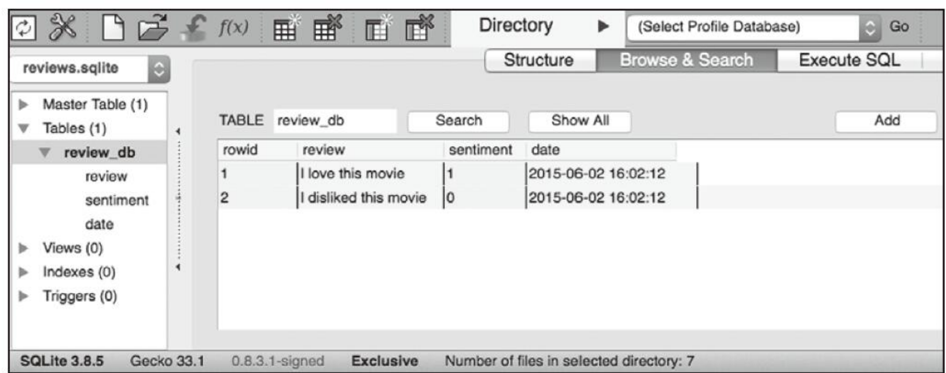
```
>>> import sqlite3
>>> import os
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute('CREATE TABLE review_db'\
...            ' (review TEXT, sentiment INTEGER, date TEXT)')
>>> example1 = 'I love this movie'
>>> c.execute("INSERT INTO review_db"\
...           " (review, sentiment, date) VALUES"\
...           " (?, ?, DATETIME('now'))", (example1, 1))
>>> example2 = 'I disliked this movie'
>>> c.execute("INSERT INTO review_db"\
...           " (review, sentiment, date) VALUES"\
...           " (?, ?, DATETIME('now'))", (example2, 0))
>>> conn.commit()
>>> conn.close()
```

在上述示例代码中，通过调用sqlite3中的connect方法，创建了一个访问SQLite数据库文件的连接（conn），当数据库文件reviews.sqlite不存在，该方法就会在movieclassifier目录下自动创建该文件。请注意，由于SQLite未实现对表的替换功能，如果要再次执行这些代码，需要读者自行在文件浏览器中删除相应的数据库文件。接下来，我们通过cursor方法创建了一个游标，就可以借助强大的SQL语句来遍历数据库中的记录。在第一次调用execute时，我们创建了一张新的数据表：review_db，并使用它存储和访问数据库中的记录。在review_db表中，我们创建了三个属性：review、sentiment，以及date，借助于它们，我们存储了两个电影评论的示例数据及其对应的类标（情感倾向）。使用SQL命令DATETIME（'now'），我们为记录增加了日期及时间戳。除了时间戳之外，我们还使用了问号（？）作为占位符，将电影评论文本（example1和example2）及其对应的类标（1和0）以元组的形式传递给execute方法。最后，我们调用commit方法保存对数据库的修改，并通过close方法关闭数据库连接。

为了检查记录是否已经正确地存储到数据库中，我们现在重建数据库连接，并通过SELECT语句获取表中自2015年1月1日之后提交的所有数据：

```
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute("SELECT * FROM review_db WHERE date"\
...     " BETWEEN '2015-01-01 00:00:00' AND DATETIME('now')")
>>> results = c.fetchall()
>>> conn.close()
>>> print(results)
[('I love this movie', 1, '2015-06-02 16:02:12'), ('I disliked this
movie', 0, '2015-06-02 16:02:12')]
```

此外，还可以使用免费的火狐浏览器插件SQLite Manager[2]，如下图所示，此插件提供了一个访问SQLite数据库的图形用户界面：



[1]　Python提供了完善的基础代码库，涵盖数据库、文本处理、文件处理、网络编程的内容，被称作内置电池。——译者注

[2]　可通过链接https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/获取。

9.3　使用Flask开发Web应用

上一节完成了用于电影评论分类的代码，现在来讨论使用Flask框架开发Web应用的基础知识。自2010年Armin Ronacher发布Flask以来，此框架获得了广泛的关注，并被流行的应用如LinkedIn及Pinterest所使用。由于Flask使用Python开发，它为Python程序员嵌入已有Python代码提供了方便的接口，在此我们将嵌入电影分类器。

Flask以微框架而著称，这意味着其内核精炼且简单，但易于通过其他库进行扩展。其学习曲线不像其他基于Python的Web框架（如Django）那样陡峭，但建议读者阅读一下Flask的官方文档：http://flask.pocoo.org/docs/0.10/，以了解其更多的功能。

如果读者的Python环境中尚未安装Flask库，请在终端窗口中通过下列命令使用pip进行安装（在本书写作过程中，最新的稳定版本为V0.10.1）：

```
pip install flask
```

9.3.1 第一个Flask Web应用

本小节中，在实现电影分类器之前，先开发一个简单的Web应用来熟悉一下Flask的API。首先，按照如下目录结构创建Web应用的框架：

```
1st_flask_app_1/
    app.py
    templates/
        first_app.html
```

app.py文件中包含为了运行Flask Web应用程序而需要在Python解释器中执行的入口代码。templates目录存放Flask用到的静态HTML文件，这些静态文件将交由浏览器进行解析。我们来看一下app.py的内容：

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('first_app.html')

if __name__ == '__main__':
    app.run()
```

在本示例下，我们以独立模块的方式运行应用程序，由此，通过参数_name_来初始化一个Flask实例，并告知Flask实例HTML模板存放路径为当前目录的templates子目录。接下来，使用路由注解（@app.route（'/'））指定触发index函数的URL路径。在本例中，index函数简单返回名为first_app.html的HTML文件，此文件存放于templates目录下。最后，使用run函数在服务器上运行程序；在确保if语句中判定条件为_name_＝＝'_main_'的情况下，此脚本可以在Python解析器中直接运

行。

现在，我们来解释一下first_app.html文件中的内容。如果读者不熟悉HTML语法，建议通过在线教程http://www.w3schools.com/html/default.asp学习HTML的基础知识。

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  </head>
  <body>
  <div>Hi, this is my first Flask web app!</div>
  </body>
</html>
```

在此，我们在HTML模板文件中插入一个div元素（块元素），此元素包含句子："Hi,this is my first Flask Web app!"。将Web应用部署到公共Web服务器上之前，Flask允许我们在本地运行应用，这对应用程序的开发和测试来说非常有用。现在，在1st_flask_app_1目录下，通过终端窗口执行下列命令启动Web应用：

```
python3 app.py
```

终端窗口中将输出：

```
* Running on http://127.0.0.1:5000/
```

此输出包含我们本地服务器的地址。我们可以在浏览器地址栏中输入该地址以查看Web程序运行效果。如果一切都运行正常，将看到一个显示如下内容的网页："Hi,this is my first Flask Web app!"。

9.3.2　表单验证及渲染

在本小节中，将使用HTML的表单升级Flask Web应用，以学习如何使用WTForms库[1]收集数据，WTForms可通过pip来安装：

```
pip install wtforms
```

Web应用会提示用户在文本框中输入自己的名字，如下图所示：

在点击提交按钮（Say Hello）后，程序将验证表单，同时返回一个新的HTML页面显示用户输入的名字：

新的应用程序所需的目录结构看起来如下所示：

```
1st_flask_app_2/
    app.py
    static/
        style.css
    templates/
        _formhelpers.html
        first_app.html
        hello.html
```

以下为修改后的app.py文件内容：

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
app = Flask(__name__)

class HelloForm(Form):
    sayhello = TextAreaField('',[validators.DataRequired()])

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)

if __name__ == '__main__':
    app.run(debug=True)
```

使用wtforms，通过TextAreaField类在起始页中嵌入文本框对index函数做了修改，TextAreaField类可以验证用户是否输入了一个有效文本。此外，我们还新增了一个hello函数，如果表单通过验证，此函数将向客户端返回HTML页面hello.html。此处，我们使用POST方式将表单数据提交给服务器。最后，在app.run()方法中设置参数debug＝True，我们启动了Flask的调试模式，这个功能在Web应用开发中非常有用。

现在，通过Jinjia2模板引擎，在_formhelpers.html文件中实现一个通用宏，后续它将被导入到first_app.html文件中用来渲染文本域：

```
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd>{{ field(**kwargs)|safe }}
  {% if field.errors %}

     <ul class=errors>
     {% for error in field.errors %}
       <li>{{ error }}</li>
     {% endfor %}
     </ul>
   {% endif %}
   </dd>
 {% endmacro %}
```

对Jinjia2模板语言更深入的讨论超出了本书的范围。不过读者可以通过http://jinja.pocoo.org找到关于Jinjia2语法的详尽文档。

接下来，我们将建立一个简单的层叠样式表（Cascading Style Sheet，CSS）文件：style.css，用来调整HTML文档页面的视觉效果。下述CSS文件能够使页面字体以2倍大小显示，该文件须存放于static子目录下，此目录是Flask存储CSS等静态文件的默认目录。代码如下：

```
body {
  font-size: 2em;
}
```

下面是修改后的first_app.html文件内容，它将显示为一个用于输入用户名字的文本框：

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  <link rel="stylesheet" href="{{ url_for('static',
    filename='style.css') }}">
  </head>
  <body>

{% from "_formhelpers.html" import render_field %}

<div>What's your name?</div>
<form method=post action="/hello">
  <dl>
    {{ render_field(form.sayhello) }}
  </dl>
  <input type=submit value='Say Hello' name='submit_btn'>
</form>
  </body>
</html>
```

我们在first_app.html标头加载了CSS文件。此时它应该可以更改HTML文件body中所有元素的大小。在HTML的body中，我们从_formhelpers.html导入了宏，并使用它来渲染app.py文件中定义的say hello表单。此外，我们还在表单中增加了一个按钮，可以让用户提交文本框中输入的内容。

在app.py中，我们定义了一个hello函数，它通过render_template（'hello.html',name＝name）返回并渲染一个HTML文件。最后，我们创建这个名为hello.html的文件，以显示用户在文本框中输入的文本。代码如下：

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  <link rel="stylesheet" href="{{ url_for('static',
    filename='style.css') }}">
```

```
    <link rel="stylesheet" href="{{ url_for('static',
      filename='style.css') }}">
    </head>
    <body>

  <div>Hello {{ name }}</div>
    </body>
  </html>
```

完成对Flask Web应用的改进后，我们可以在app所在目录下，通过执行下述命令在本地运行此程序，并在浏览器中通过http://127.0.0.1:5000/来查看运行结果：

```
python3 app.py
```

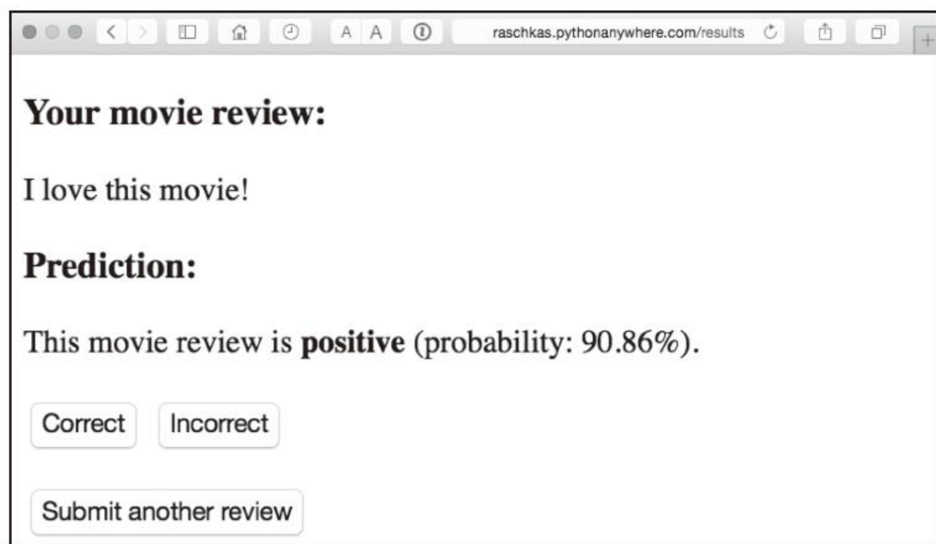
对初涉Web开发的读者来说，某些概念乍看好像非常复杂。若遇到这种情况，建议读者先在某个目录中设置前面提及的文件并仔细检查。你会发现Flask Web框架其实非常易用，也不像最初看上去那样复杂。关于Flask更多的帮助信息，请访问其出色的在线文档和示例：http://flask.pocoo.org/docs/0.10/。

[1] https://wtforms.readthedocs.org/en/latest/.

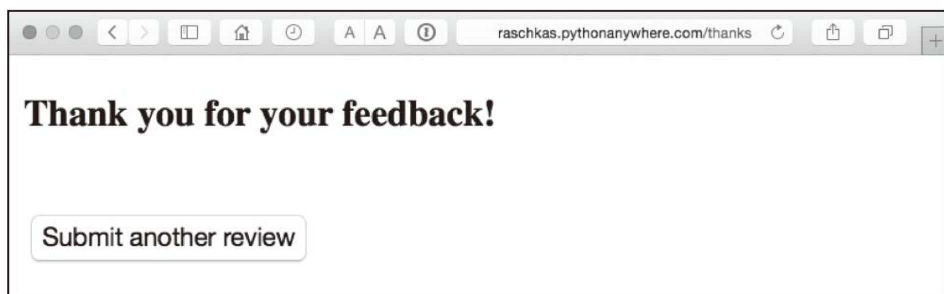
9.4 将电影分类器嵌入Web应用

目前，我们对使用Flask进行Web开发有了一定的认识，下面更进一步，将电影分类器嵌入到Web应用中。在本节，我们先开发一个Web应用，此应用会提示用户输入一个电影评论，如下图所示：

在评论提交后，会返回一个新的页面，页面中会显示预测类标及评论属于此类别的概率。此外，用户还可以使用“正确”（Correct）或者“不正确”（Incorrect）两个按钮对预测结果做出反馈，如下图所示：



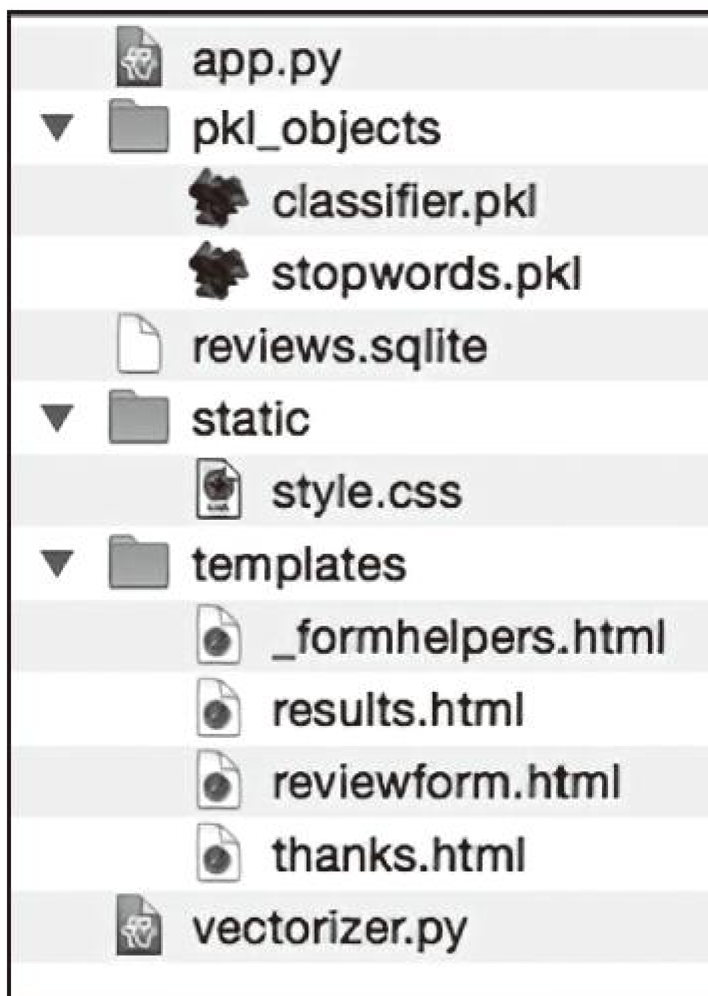
如果用户点击了“正确”或者“不正确”按钮，分类模型将基于用户的反馈进行更新。此外，对于用户输入的评论文本，以及从按钮点击中推测出的类标等信息，我们将使用SQLite数据库进行保存，为将来的预测提供参考。用户点击反馈按钮后看到的第三个页面，显示了简单的感谢信息和一个“提交其他评论”（Submit another review）按钮，此按钮将页面重定向到起始页面。如下图所示：



在详细学习此Web应用的代码之前，建议读者通过http://raschkas.pythonanywhere.com了解一下此应用的示例，以对本小节待讲

解内容有个更好的认识。

从全局角度出发，我们先看一下此电影评论分类应用的目录结构，如下图所示：



在本章前面的小节中，我们已经创建了vecorizer.py文件、SQLite数据库reviews.sqlite，以及可以保存经过序列化的Python对象的pkl_objects子

目录。

主目录下的app.py文件是包含Flask代码的Python脚本，在Web应用中，我们将使用review.sqlite数据库文件（本章前面小节中创建的文件）保存用户提交的电影评论数据。templates子目录存储将被Flask渲染的HTML模板文件；static子目录仅包含一个简单的CSS文件，用于调整HTML页面渲染效果。

由于app.py文件较长，我们分两步来分析。首先导入所需的Python模块及对象，并通过反序列化恢复我们的分类模型：

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
import pickle
import sqlite3
import os
import numpy as np
# import HashingVectorizer from local dir
```

```
from vectorizer import vect

app = Flask(__name__)

######## Preparing the Classifier
cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                 'pkl_objects/classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

def classify(document):
    label = {0: 'negative', 1: 'positive'}
    X = vect.transform([document])
    y = clf.predict(X)[0]
    proba = np.max(clf.predict_proba(X))
    return label[y], proba

def train(document, y):
    X = vect.transform([document])
    clf.partial_fit(X, [y])

def sqlite_entry(path, document, y):
    conn = sqlite3.connect(path)
    c = conn.cursor()
    c.execute("INSERT INTO review_db (review, sentiment, date)"\
    " VALUES (?, ?, DATETIME('now'))", (document, y))
    conn.commit()
    conn.close()
```

我们对app.py的第一部分内容应该很熟悉了。首先导入了HashingVectorizer并对逻辑斯谛回归分类器进行反序列化操作。接下来，定义了一个classify函数返回针对给定文本文档的预测类标及其对应的概率。当文档及其对应类标可用时[1]，可使用train函数更新分类器模型。使用sqlite_entry函数，我们可以将用户输入的评论信息及其对应的类标，以及时间戳等存入SQLite数据库。请注意，如果重启了Web应用，那么clf对象将被重置为其原始的序列化状态。在本章最后，读者将学到如何使用SQLite数据库收集到的数据对分类器进行永久更新。

app.py第二部分的内容看起来也非常熟悉：

```
app = Flask(__name__)
class ReviewForm(Form):
    moviereview = TextAreaField('',
                                [validators.DataRequired(),
                               validators.length(min=15)])

@app.route('/')
def index():
    form = ReviewForm(request.form)
    return render_template('reviewform.html', form=form)

@app.route('/results', methods=['POST'])
def results():


    form = ReviewForm(request.form)
    if request.method == 'POST' and form.validate():
        review = request.form['moviereview']
        y, proba = classify(review)
        return render_template('results.html',
                                content=review,
                                prediction=y,
                                probability=round(proba*100, 2))
    return render_template('reviewform.html', form=form)

@app.route('/thanks', methods=['POST'])
def feedback():
    feedback = request.form['feedback_button']
    review = request.form['review']
    prediction = request.form['prediction']

    inv_label = {'negative': 0, 'positive': 1}
    y = inv_label[prediction]
    if feedback == 'Incorrect':
        y = int(not(y))
    train(review, y)
    sqlite_entry(db, review, y)
    return render_template('thanks.html')

if __name__ == '__main__':
    app.run(debug=True)
```

我们定义了一个ReviewForm类来实例化TextAreaField对象，它将在模版文件review-form.html（Web应用的起始页）中被渲染。此对象同样也会被index函数所渲染。以validators.length（min＝15）为参数，可以限制用户的输入至少为15个字符。在result函数内，我们能够获取到用户

在Web表单中提交的内容，并将其传递给电影评论分类器以预测情感倾向，预测结果将显示在渲染后的results.html模板中。

乍一看，feedback函数好像有些复杂。如果用户在results.html页面点击了“正确”或“不正确”反馈按钮，此函数将分析出正确的类标，并将正确的情感预测转换成整数类标，以便使用train函数更新分类器，train函数在app.py脚本的第一部分已经做了介绍。此外，如果用户提交了反馈，sqlite_entry函数就会在SQLite数据库中插入一条新的数据，最后thanks.html模板就会被渲染出来以感谢用户的反馈。

接下来，看一下reviewform.html模板，它是Web应用的起始页面：

```
<!doctype html>
<html>
<head>
    <title>Movie Classification</title>
</head>
  <body>

<h2>Please enter your movie review:</h2>

{% from "_formhelpers.html" import render_field %}

<form method=post action="/results">
  <dl>

    {{ render_field(form.moviereview, cols='30', rows='10') }}
  </dl>
  <div>
    <input type=submit value='Submit review' name='submit_btn'>
  </div>
</form>

  </body>
</html>
```

在此，我们导入了本章的9.3.2节中定义的_formhelpers.html模板。宏

中的render_field函数用于渲染TextAreaField对象，此对象用于收集用户填写的电影评论信息，并可通过页面底部的“提交评论”（Submit review）按钮将信息提交到服务器。此TextAreaField对象的宽度为30个列宽，高度为10行。

下一个模板，results.html，看上去很有趣：

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
  <link rel="stylesheet" href="{{ url_for('static',
    filename='style.css') }}">
  </head>
  <body>

<h3>Your movie review:</h3>
<div>{{ content }}</div>

<h3>Prediction:</h3>
<div>This movie review is <strong>{{ prediction }}</strong>
  (probability: {{ probability }}%).</div>

<div id='button'>
  <form action="/thanks" method="post">
    <input type=submit value='Correct' name='feedback_button'>
    <input type=submit value='Incorrect' name='feedback_button'>
    <input type=hidden value='{{ prediction }}' name='prediction'>
    <input type=hidden value='{{ content }}' name='review'>
  </form>
</div>

<div id='button'>
  <form action="/">
    <input type=submit value='Submit another review'>
  </form>
</div>

  </body>
</html>
```

首先，我们通过{{content}}、{{prediction}}，以及{{probability}}分别插入了用户提交的评论、预测类标，以及对应的概率。读者可能注意到了，我们在包含“正确”及“不正确”按钮的表单中第二次使用了

{{content}}和{{prediction}}占位符。这种旨在当用户点击了任一按钮后，以POST方式将数据返回给服务器，以便进行更新模型和存储评论数据。此外，我们在results.html文件的头部导入了CSS文件（style.css）。此文件的设置非常简单：它将Web应用中内容的宽度设置为600像素，并使用＜div id＝“button”＞将“正确”与“不正确”按钮下移了20像素：

```
body{
  width:600px;
}
#button{
  padding-top: 20px;
}
```

此CSS文件仅相当于一个占位符，读者可以根据自己的喜好任意调整页面的显示方式。

Web应用要实现的最后一个模板页面是thanks.html。顾名思义，此页面以友好的方式在用户点击“正确”或“不正确”按钮给出反馈后表示了感谢。此外，在页面的底部，我们给出了一个“提交其他评论”的按钮，它可以将用户重定向到起始页。thanks.html的内容如下：

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
</head>
  <body>

<h3>Thank you for your feedback!</h3>
<div id='button'>
  <form action="/">
    <input type=submit value='Submit another review'>
  </form>
</div>

  </body>
</html>
```

在进入下一小节并将应用部署到公共Web服务器上之前，我们不妨在终端窗口中输入如下命令，从而在本地环境中启动Web应用：

```
python3 app.py
```

完成Web应用的测试后，不要忘记将app.py脚本中app.run()命令的debug＝True参数去掉。

[1] 即用户输入文档，并对预测结果做了相关反馈时。——译者注

9.5　在公共服务器上部署Web应用

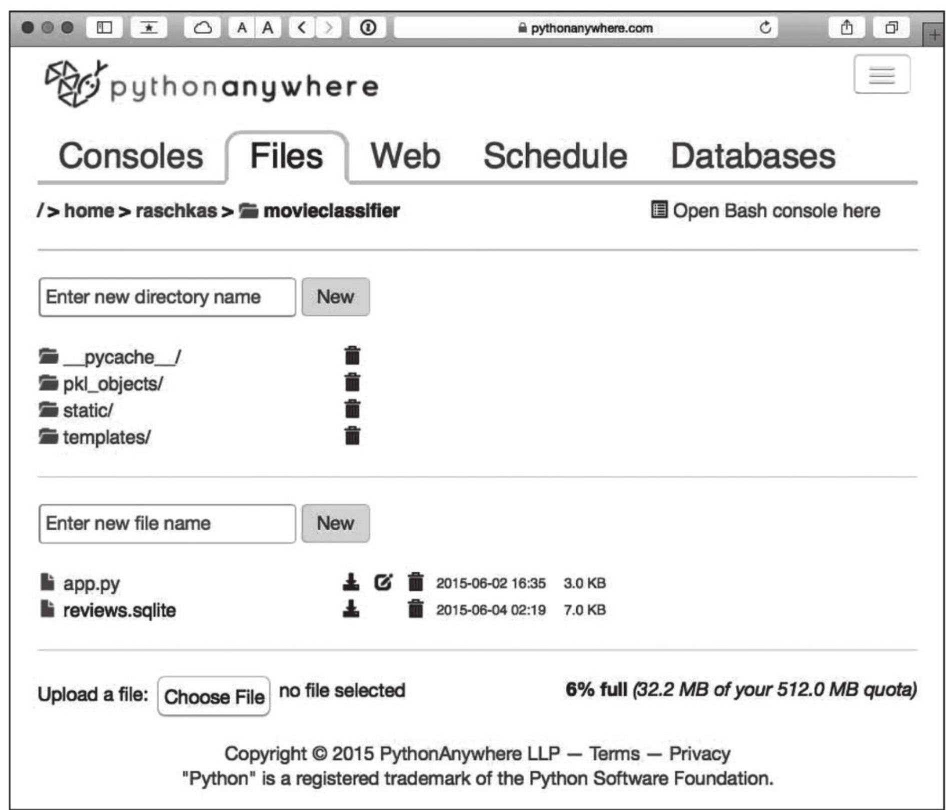
当完成应用的本地测试后，我们现在可以将其部署到公共Web服务器上了。在本书中，我们将使用PythonAnywhere托管Web服务，它专门用于Python Web应用的托管。此外，PythonAnywhere还提供了初学者账户，我们可以免费运行一个Web应用程序。

要创建一个新的PythonAnywhere账户，我们需要访问https://www.pythonanywhere.com，并点击右上角的“Pricing&signup”链接。接下来，点击“Create a Beginner account”按钮，此时需要输入用户名、密码，及一个有效的电子邮件地址，阅读并同意相关条款之后，我们就拥有一个新的账户了。

不过，免费账户无法在命令行终端通过SSH协议访问远程服务器。因此，只能使用PythonAnywhere Web界面管理我们的Web应用。在上传Web应用之前，需要事先在PythonAnywhere账户中创建一个新的Web应用。点击右上角的“Dashboard”按钮，就可以看到页面顶部显示的控制面板。接下来，点击页面顶部的“Web”标签，然后点击左侧的“Add a new Web app”按钮创建一个新的基于Python 3.4的Flask Web应用，并将其命名为movieclassifier。

在PythonAnywhere中创建好新的应用后，我们选择页面顶部的File标签，使用PythonAnywhere的Web界面上传本地movieclassifier目录下的文件。完成本地计算机中Web应用文件的上传后，在PythonAnywhere账号中能够看到一个movieclassifier目录。它与我们本地计算机上

movieclassifier目录中的文件和目录结构完全相同，如下图所示：



最后，我们再次选择“Web”标签，并点击“Reload＜读者自己注册的用户名＞.pytho-nanywhere.com”按钮发送更新，刷新Web应用。这时，Web应用应该已经启动和运行起来了，可访问＜读者自己注册的用户名＞.pythonanywhere.com。

不幸的是，Web服务器可能对Web应用程序中的细微问题也相当敏感。如果读者通过浏览器访问在PythonAnywhere中的应用程序时出错，可以在个人账号主页面的“Web”标签下检查服务器日志信息，以更好

地帮助查找问题所在。

更新电影评论分类器

当收到用户关于分类的反馈后，模型会自动即时更新，但是如果服务器崩溃或者重启，clfd对象的更新就会被重置。如果我们重新加载Web应用，clf对象将通过classifier.pkl文件重新初始化。使得更新能够持久保存的一个方法就是：模型一旦被更新就立刻序列化新的clf对象。但是，随着用户的增多，此方案效率会逐渐低下，而且如果用户同时提交反馈信息，有可能会损坏序列化文件。另一种解决方案就是使用SQLite数据库保存的反馈信息更新预测模型。我们可以从PythonAnywhere的服务器上下载SQLite数据库，在本地计算机上更新clf对象，并上传新的序列化文件到PythonAnywhere。为了在本地计算机上更新分类器，我们在movieclassifier目录下创建一个update.py脚本文件，并键入以下代码：

```
import pickle
import sqlite3
import numpy as np
import os

# import HashingVectorizer from local dir
from vectorizer import vect

def update_model(db_path, model, batch_size=10000):

    conn = sqlite3.connect(db_path)
    c = conn.cursor()
    c.execute('SELECT * from review_db')

    results = c.fetchmany(batch_size)
    while results:
        data = np.array(results)
        X = data[:, 0]
        y = data[:, 1].astype(int)

        classes = np.array([0, 1])
        X_train = vect.transform(X)
        clf.partial_fit(X_train, y, classes=classes)
        results = c.fetchmany(batch_size)

    conn.close()
    return None
cur_dir = os.path.dirname(__file__)

clf = pickle.load(open(os.path.join(cur_dir,
                 'pkl_objects',
                 'classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

update_model(db_path=db, model=clf, batch_size=10000)

# Uncomment the following lines if you are sure that
# you want to update your classifier.pkl file
# permanently.

# pickle.dump(clf, open(os.path.join(cur_dir,
#              'pkl_objects', 'classifier.pkl'), 'wb')
#              , protocol=4)
```

update_model函数以每次10000条记录的方式批量从数据库中读取数据，如果记录数量小于10000条，则根据实际数量读取。或者我们可以将代码中的fetchmany改为fetchone，从而每次只读取一条记录，但这样计算效率会很差。如果使用fetchall方法，则可能在面对海量数据时导致计算

机或服务器的内存溢出。

创建好update.py脚本后，我们将其上传到PythonAnywhere服务器上的movieclassifier目录下，并在应用的入口脚本app.py中导入update_model函数，确保每次重启Web服务器后，能够根据SQLite数据库内容对分类器进行更新。为了实现此功能，我们需要在app.py开头增加一行导入update.py脚本中update_model函数的代码：

```
# import update function from local dir
from update import update_model
```

然后，我们就可以在应用程序的主脚本中调用update_model函数了：

```
…
if __name__ == '__main__':
    update_model(filepath=db, model=clf, batch_size=10000)
…
```

本章小结

在本章，读者学到了许多对扩充我们机器学习理论知识的实用主题。我们学习了如何在模型训练完成后对其进行持久化化，以及如何在日后重新使用经过持久化的模型。此外，我们使用SQLite数据库高效地存储数据，并创建了一个Web应用，使得我们电影分类器可以供外界使用。

在本书中，我们已经讨论了许多机器学习的概念、最佳实践方式，以及用于分类的监督模型。在下一章，我们将学习监督学习的另一个分支：回归分析，与我们已经学习过的分类模型不同，它的输出不是基于类别数据的类标，而是在某个区间上的连续值。

第10章　使用回归分析预测连续型目标变量

在前面的章节中，我们学习了许多关于监督学习的概念，并且针对分类问题训练了许多不同的模型来预测成员分组或样本的类别归属。本章我们将进入监督学习的另一个分支：回归分析（regression analysis）。

回归模型（regression model）可用于连续型目标变量的预测分析，这使得它在探寻变量间关系、评估趋势、作出预测等科学及工业领域应用中极具吸引力。具体的例子如：预测公司在未来几个月的销售情况等。

在本章，我们将介绍回归模型的主要概念，将涵盖如下主题：

·数据集的探索与可视化

·实现线性回归模型的不同方法

·训练可处理异常值的回归模型

·回归模型的评估及常见问题

·基于非线性数据拟合回归模型

10.1 简单线性回归模型初探

简单（单变量）线性回归的目标是：通过模型来描述某一特征（解释变量x）与连续输出（目标变量y）之间的关系。当只有一个解释变量时，线性模型的函数定义如下：

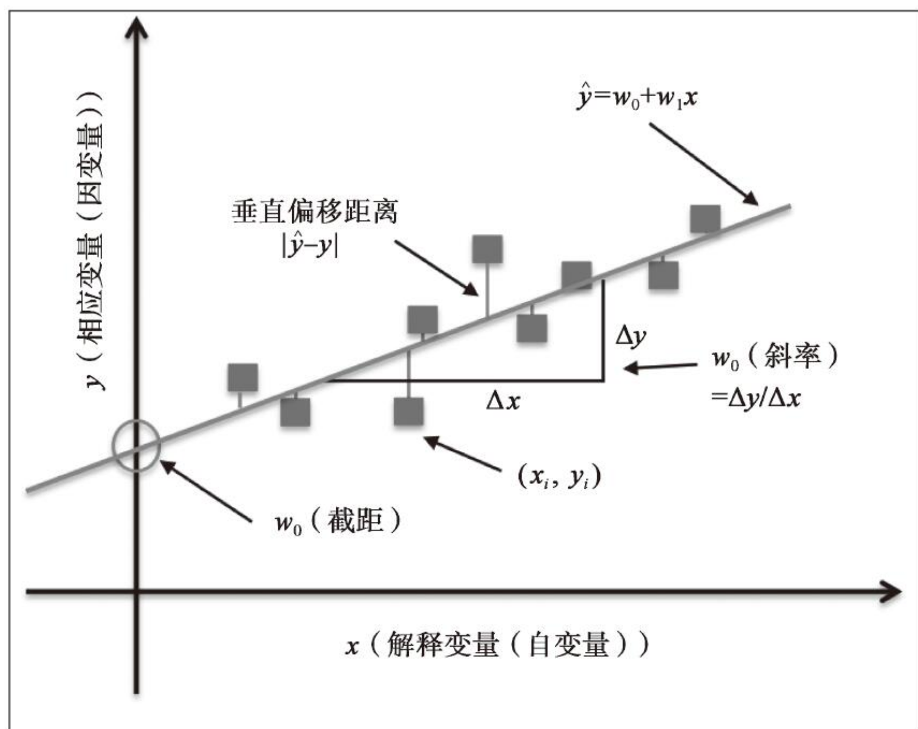
$$y = w_0 + w_1 x$$

其中，权值w_0为函数在y轴上的截距，w_1为解释变量的系数。我们的目标是通过学习得到线性方程的这两个权值，并用它们描述解释变量与目标变量之间的关系，当解释变量为非训练数据集中数据时，可用此线性关系来预测对应的输出。

基于前面所定义的线性方程，线性回归可看作是求解样本点的最佳拟合直线，如下图所示。

这条最佳拟合线也被称为回归线（regression line），回归线与样本点之间的垂直连线即所谓的偏移（offset）或残差（residual）——预测的误差。

在只有一个解释变量的特殊情况下，线性回归也称为简单线性回归（simple linear regression），当然，我们可以将线性回归模型扩展为多个解释变量。此时，即为所谓的多元线性回归（multiple linear regression）：



$$y = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \sum_{i=0}^{n} w_ix_i = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$$

其中，w_0为$x_0 = 1$时在y轴上的截距。

10.2 波士顿房屋数据集

在实现第一个线性回归模型之前，先介绍一个新的数据集：波士顿房屋数据集（Housing Dataset），它是由D.Harrison和D.L.Rubinfeld于1978年收集的波士顿郊区房屋的信息。此房屋数据集可免费使用，读者可在UCI机器学习数据库中下载：https://archive.ics.uci.edu/ml/datasets/Housing。

数据集中506个样本的特征描述如下：

·CRIM：房屋所在镇的犯罪率。

·ZN：用地面积远大于25000平方英尺[1]的住宅所占比例。

·INDUS：房屋所在镇无零售业务区域所占比例。

·CHAS：与查尔斯河有关的虚拟变量（如果房屋位于河边则值为1，否则为0）。

·NOX：一氧化氮浓度（每千万分之一）。

·RM：每处寓所的平均房间数。

·AGE：业主自住房屋中，建于1940年之前的房屋所占比例。

·DIS：房屋距离波士顿五大就业中心的加权距离。

·RAD：距离房屋最近公路入口编号。

·TAX：每一万美元全额财产税金额。

·PTRATIO：房屋所在镇的师生比。

·B：计算公式为1000（Bk - 0.63）^2，其中Bk为房屋所在镇非裔美籍人口所占比例。

·LSTAT：弱势群体人口所占比例。

·MEDV：业主自住房屋的平均价格（以1000美元为单位）。

在本章的后续内容中，我们将以房屋价格（MEDV）作为目标变量——使用其他13个变量中的一个或多个值作为解释变量对其进行预测。在进一步研究此数据集之前，我们先从UCI库中把它导入到pandas的DataFrame中：

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/housing/housing.data',
...                   header=None, sep='\s+')
>>> df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
...               'NOX', 'RM', 'AGE', 'DIS', 'RAD',
...               'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
>>> df.head()
```

为了确保数据已经正确加载，我们先显示数据集的前5行，如下图所示：

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

可视化数据集的重要特征

探索性数据分析（Exploratory Data Analysis，EDA）是机器学习模型训练之前的一个重要步骤。在本节的后续内容中，借助EDA图形工具箱中那些简单且有效的技术，可以帮助我们直观地发现数据中的异常情况、数据的分布情况，以及特征间的相互关系。

首先，借助散点图矩阵，我们以可视化的方法汇总显示各不同特征两两之间的关系。为了绘制散点图矩阵，我们需要用到seaborn库中的pairplot函数（http://stanford.edu/~mwaskom/software/seaborn/），它是在matplotlib基础上绘制统计图像的Python库。

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> sns.set(style='whitegrid', context='notebook')
>>> cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
>>> sns.pairplot(df[cols], size=2.5);
>>> plt.show()
```

如下图所示，散点图矩阵以图形的方式对数据集中特征间关系进行了描述：

导入seaborn库后，会覆盖当前Python会话中matplotlib默认的图像显示方式。如果读者不希望使用seaborn风格的设置，可以通过如下命令重设为matplotlib的风格：

```
>>> sns.reset_orig()
```

出于篇幅限制及可读性的考虑，我们仅绘制了数据集中的5列：LSTAT、INDUS、NOX、RM和MEDV。不过，建议读者绘制整个DataFrame的散点图矩阵，以对数据做进一步的探索。

通过此散点图矩阵，我们可以快速了解数据是如何分布的，以及其中是否包含异常值。例如，我们可直观看出RM和房屋价格MEDV（第5列和第4行）之间存在线性关系。此外，从MEDV直方图（散点图矩阵的右下角子图）中可以发现：MEDV看似呈正态分布，但包含几个异常值。

请注意，不同于人们通常的理解，训练一个线性回归模型并不需要解释数量或者目标变量呈正态分布。正态假设仅适用于某些统计检验和假设检验（Montgomery,D.C.,Peck,E.A.,and Vining,G.G.Introduction to linear regression analysis.John Wiley and Sons,2012,pp.318-319），这

些内容已经超出了本书讲解的范围，有兴趣的读者可以根据自身情况进行学习。

为了量化特征之间的关系，我们创建一个相关系数矩阵。相关系数矩阵与我们第5章“主成分分析”一节中讨论过的协方差矩阵是密切相关的。直观上来看，我们可以把相关系数矩阵看作协方差矩阵的标准化版本。实际上，相关系数矩阵就是在将数据标准化后得到的协方差矩阵。

相关系数矩阵是一个包含皮尔逊积矩相关系数（Pearson product-moment correlation coefficient，通常记为Pearson екr）的方阵，它用来衡量两两特征间的线性依赖关系。相关系数的取值范围为-1到1。如果r = 1，代表两个特征完全正相关；如果r = 0，则不存在相关关系；如果r = -1，则两个特征完全负相关。如前所述，皮尔逊相关系数可用两个特征x和y间的协方差（分子）除以它们标准差的乘积（分母）来计算。

$$r=\frac{\sum_{i=1}^{n}\left[\left(x^{(i)}-\mu_x\right)\left(y^{(i)}-\mu_y\right)\right]}{\sqrt{\sum_{i=1}^{n}\left(x^{(i)}-\mu_x\right)^2}\sqrt{\sum_{i=1}^{n}\left(y^{(i)}-\mu_y\right)^2}}=\frac{\sigma_{xy}}{\sigma_x\sigma_y}$$

其中，μ为样本对应特征的均值，σ_{xy}为特征x和y间的协方差，而σ_x和σ_y分别为两个特征的标准差。

可以证明：经标准化各特征间的协方差实际上等价于它们的线性相关系数。

首先对特征x和y做标准化处理，得到它们的z分数（z-score），并分别记为x'和y'：

$$x'=\frac{x-\mu_x}{\sigma_x}, \quad y'=\frac{y-\mu_y}{\sigma_y}$$

我们曾经使用下面的公式计算过两个特征（人口）间的协方差：

$$\sigma_{xy}=\frac{1}{n}\sum_{i}^{n}\left[\left(x^{(i)}-\mu_x\right)\left(y^{(i)}-\mu_y\right)\right]$$

由于特征经标准化后，其均值为0，由此，可通过下式来计算特征经缩放后的协方差：

$$\sigma'_{xy}=\frac{1}{n}\sum_{i}^{n}\left(x'-0\right)\left(y'-0\right)$$

将x'和y'带入后，可得：

$$\frac{1}{n}\sum_{i}^{n}\left(\frac{x-\mu_x}{\sigma_x}\right)\left(\frac{y-\mu_y}{\sigma_y}\right)$$

$$\frac{1}{n\cdot\sigma_x\sigma_y}\sum_{i}^{n}\left(x^{(i)}-\mu_x\right)\left(y^{(i)}-\mu_y\right)$$

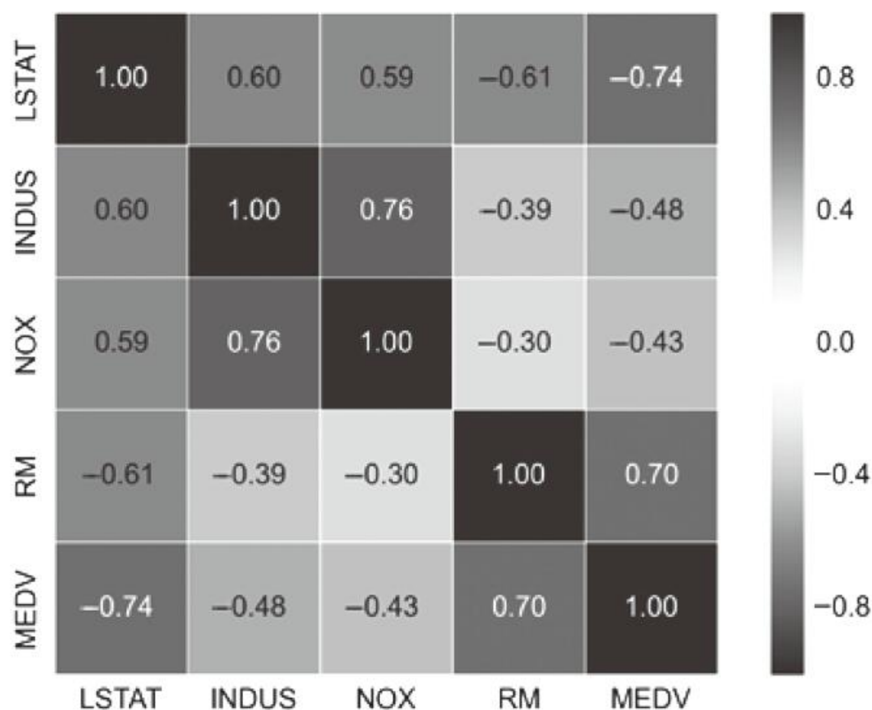
上式可简写为：

$$\sigma'_{xy}=\frac{\sigma_{xy}}{\sigma_x\sigma_y}$$

在下面的示例代码中，将使用NumPy的corrcoef函数计算前面散点图矩阵中5个特征间的相关系数矩阵，并使用seaborn的heatmap函数绘制其对应的热度图：

```
>>> import numpy as np
>>> cm = np.corrcoef(df[cols].values.T)
>>> sns.set(font_scale=1.5)
>>> hm = sns.heatmap(cm,
...                  cbar=True,
...                  annot=True,
...                  square=True,
...                  fmt='.2f',
...                  annot_kws={'size': 15},
...                  yticklabels=cols,
...                  xticklabels=cols)
>>> plt.show()
```

从结果图像中可见，相关系数矩阵为我们提供了另外一种有用的图形化数据描述方式，由此可以根据各特征间的线性相关性进行特征选择：



为了拟合线性回归模型，我们主要关注那些跟目标变量MEDV高度相关的特征。观察前面的相关系数矩阵，可以发现MEDV与变量LSTAT的相

关性最大（-0.74）。大家应该还记得，前面的散点图矩阵显示LSTAT和MEDV之间存在明显的非线性关系。另一方面，正如散点图矩阵所示，RM和MEDV间的相关性也较高（0.70），考虑到从散点图中观察到了这两个变量之间的线性关系，因此，在后续小节，中使用RM作为解释变量进行简单线性回归模型训练，是一个较好的选择。

[1] 1平方英尺≈0.093平方米。

10.3 基于最小二乘法构建线性回归模型

在本章开始时讨论过，可将线性回归模型看作通过训练数据的样本点来寻找一条最佳拟合直线。不过，在此既没有对最佳拟合做出定义，也没有研究拟合类似模型的各种技术。在接下来的小节中，我们将消除读者对上述问题的疑惑：通过最小二乘法（Ordinary Least Squares，OLS）估计回归曲线的参数，使得回归曲线到样本点垂直距离（残差或误差）的平方和最小。

10.3.1　通过梯度下降计算回归参数

回顾一下第2章中介绍的自适应线性神经元（Adaptive Linear Neuron，Adaline）：人工神经元中使用了一个线性激励函数，同时还定义了一个代价函数J（·），通过梯度下降（Gradient Descent，GD）、随机梯度下降（Stochastic Gradient Descent，GD）等优化算法使得代价函数最小，从而得到相应的权重。Adaline中的代价函数就是误差平方和（Sum of Squared Error，SSE），它等同于我们定义的OLS代价函数：

$$J(w)=\frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)}-\hat{y}^{(i)}\right)^2$$

其中，$\hat{y}$为通过$\hat{y}=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$得到的预测值（注意，此处的系数1/2仅是为了方便推导GD更新规则）。本质上，OLS线性回归可以理解为无单位阶跃函数的Adaline，这样我们得到的是连续型的输出值，而不是以-1和1来代表的类标。为了说明两者的相似程度，使用第2章中实现的GD方法，并且移除其单位阶跃函数来实现我们第一个线性回归模型：

```
class LinearRegressionGD(object):

    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return self.net_input(X)
```

如果读者需要补习权重更新的内容——为何沿着与梯度相反的方向前进一步，请再复习第2章中的相关内容。

为了在实践中熟悉LinearRegressionGD类，我们使用了房屋数据集中的RM（房间数量）作为解释变量来训练模型以预测MEDV（房屋价格）。此外，为了使得梯度下降算法收敛性更佳，在此对相关变量做了标准化处理，代码如下：

```
>>> X = df[['RM']].values
>>> y = df['MEDV'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y)
>>> lr = LinearRegressionGD()
>>> lr.fit(X_std, y_std)
```

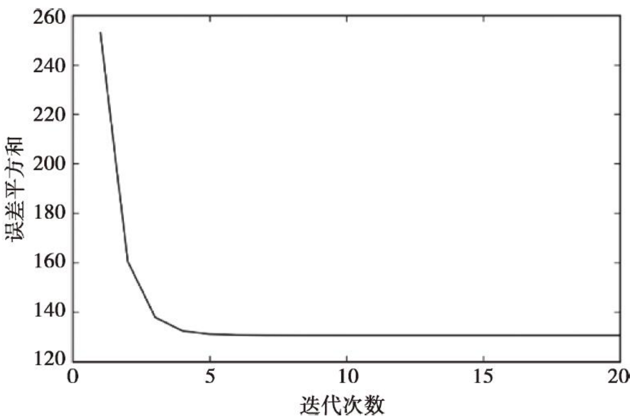
第2章中曾介绍过，梯度下降算法能够进行收敛性检查，使用这类优

化算法时，将代价看作迭代次数的函数（基于训练数据集），并将其绘制成图是个非常好的做法。简而言之，我们将再次绘制迭代次数对应的代价函数的值，以检查线性回归是否收敛：

```
>>> plt.plot(range(1, lr.n_iter+1), lr.cost_)
>>> plt.ylabel('SSE')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

从下图中可以看到，经过第5次迭代后，GD算法收敛了：

接下来，我们将线性回归曲线与训练数据拟合情况绘制成图。为达到此目的，我们定义了一个辅助函数用来绘制训练样本的散点图，同时绘制出相应的回归曲线：

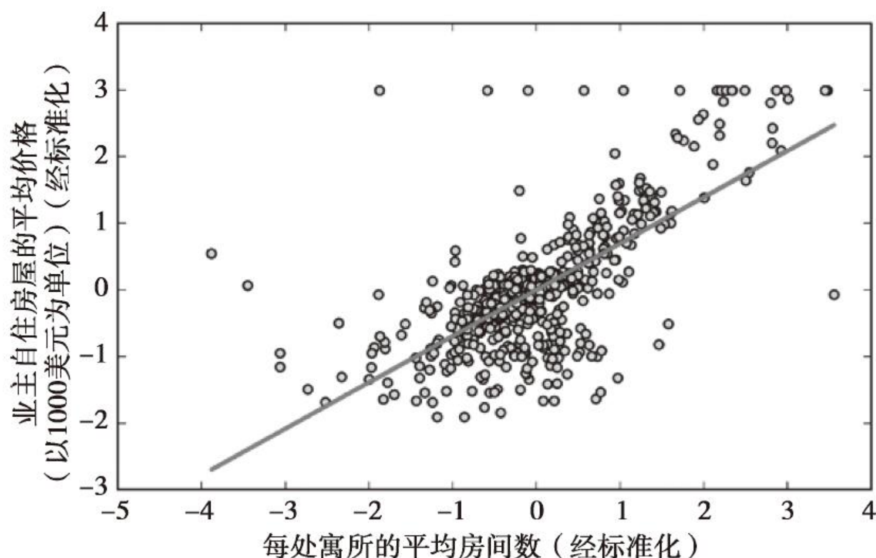



```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='blue')
...     plt.plot(X, model.predict(X), color='red')
...     return None
```

至此，我们使用lin_regplot函数绘制房间数与房屋价格之间的关系图：

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel('Average number of rooms [RM] (standardized)')
>>> plt.ylabel('Price in $1000\'s [MEDV] (standardized)')
>>> plt.show()
```

正如下图所示，线性回归曲线反映出了一般趋势，随着房间数量的增加，房价呈增长趋势：



上述结论是基于直觉观察得到的，但是数据也同样告诉我们，房间数在很多情况下并不能很好地解释房价。本章后续内容将讨论如何量化回归模型的性能。有趣的是，我们观察到一条奇怪的直线y = 3，这意味着房价被限定了上界。在某些应用中，给出变量在原始取值区间上的预测值也是非常重要的。为了将预测价格缩放到以1000美元为价格单位的坐标轴上，我们使用了StandardScaler的inverse_transform方法：

```
>>> num_rooms_std = sc_x.transform([5.0])
>>> price_std = lr.predict(num_rooms_std)
>>> print("Price in $1000's: %.3f" % \
...        sc_y.inverse_transform(price_std))
Price in $1000's: 10.840
```

上述代码中，我们使用了之前训练好的线性回归模型来预测带有5个房间的房屋价格。根据我们模型的预测，此房屋价值10840美元。

值得一提的是：对于经过标准化处理的变量，我们无需更新其截距的权重，因为它们在y轴上的截距始终为0。我们可以通过输出其权重来快速确认这一点：

```
>>> print('Slope: %.3f' % lr.w_[1])
Slope: 0.695
>>> print('Intercept: %.3f' % lr.w_[0])
Intercept: -0.000
```

10.3.2 使用scikit-learn估计回归模型的系数

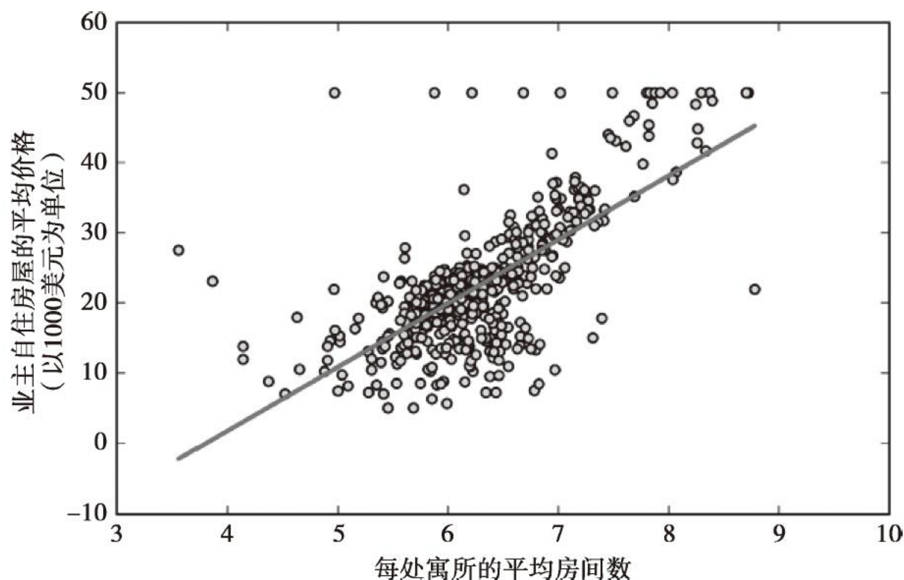
上一小节中，我们实现了一个可用的回归分析模型。不过在实际应用中，我们可能会更关注如何高效地实现模型，例如，scikit-learn中的LinearRegression对象使用了LIBLINEAR库以及先进的优化算法，可以更好地使用经过标准化处理的变量。这正是特定应用所需要的：

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> print('Slope: %.3f' % slr.coef_[0])
Slope: 9.102
>>> print('Intercept: %.3f' % slr.intercept_)
Intercept: -34.671
```

执行上述代码后可见，使用经过标准化处理的RM和MEDV数据拟合scikit-learn中的LinearRegression模型得到了不同的模型系数。我绘制出MEDV与RM之间的关系图，并与GD算法实现的模型进行比较：

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Average number of rooms [RM] (standardized)')
>>> plt.ylabel('Price in $1000\'s [MEDV] (standardized)')
>>> plt.show()
```

上述代码绘制出了训练数据和拟合模型的图像，从图中可看出，总体结果与GD算法实现的模型是一致的：



在大多数介绍统计科学的教科书中，都可以找到使用最小二乘法求解线性方程组的封闭方法：

$$w_1 = (X^T X)^{-1} X^T y$$
$$w_0 = \mu_y - \mu_{\hat{y}} \mu_{\hat{y}}$$

其中，μy为真实目标值的均值，为经预测得到目标值的均值。

这种方法的优点在于：它一定能够分析找到最优解。不过，如果要处理的数据集量很大，公式中逆矩阵的计算成本会非常高，或者矩阵本身为奇异矩阵（不可逆），这就是在特定情况下我们更倾向于使用交互式方法的原因。

若读者对如何得到正规方程感兴趣，建议阅读Stephen Pollock博士在英国莱斯特大学演讲讲义中的章节“The Classical Linear Regression Model”，可通过链接http://www.le.ac.uk/users/dsgp1/COURSES/

MESOMET/ECMETXT/06mesmet.pdf获取。

10.4　使用RANSAC拟合高鲁棒性回归模型

异常值对线性回归模型具有严重的影响。某些情况下，数据集的一个非常小的子集也可能会对模型参数的估计造成很大的影响。目前已有许多统计检测方法能够检测异常值，但该部分内容已超出本书范围。不过，作为数据方面的研究人员，我们需要根据相关领域的专业知识，并结合自身的判断清除异常值。

作为清除异常值的一种高鲁棒性回归方法，在此我们将学习随机抽样一致性（RANdom SAmple Consensus，RANSAC）算法，使用数据的一个子集（即所谓的内点，inlier）来进行回归模型的拟合。

我们将RANSAC算法的工作流程总结如下：

1）从数据集中随机抽取样本构建内点集合来拟合模型。

2）使用剩余数据对上一步得到的模型进行测试，并将落在预定公差范围内的样本点增至内点集合中。

3）使用全部的内点集合数据再次进行模型的拟合。

4）使用内点集合来估计模型的误差。

5）如果模型性能达到了用户设定的特定阈值或者迭代达到了预定次数，则算法终止，否则跳转到第1步。

现在使用scikit-learn的RANSACRegressor对象来实现我们的线性模

型：

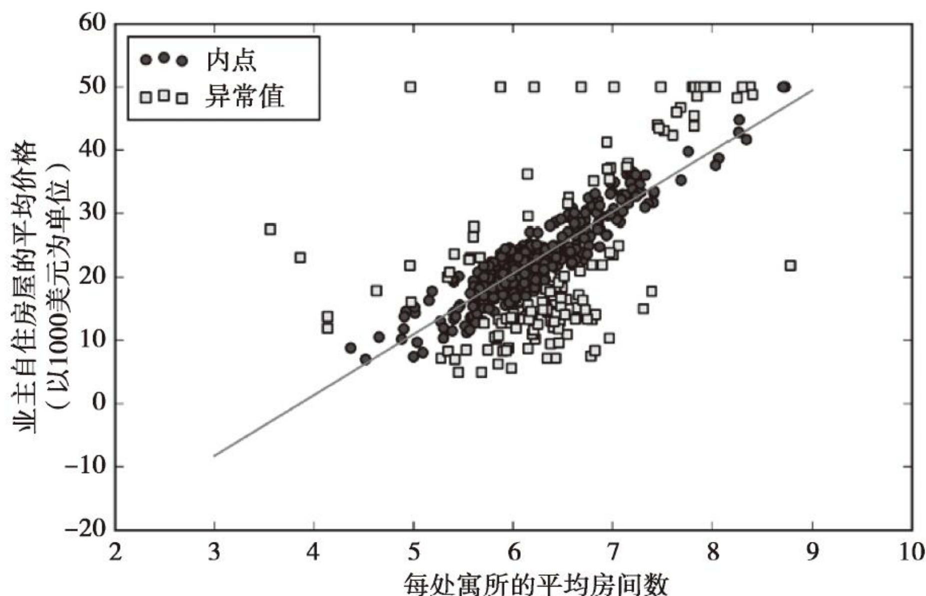
```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(LinearRegression(),
...                  max_trials=100,
...                  min_samples=50,
...                  residual_metric=lambda x: np.sum(np.abs(x), axis=1),
...                  residual_threshold=5.0,
...                  random_state=0)
>>> ransac.fit(X, y)
```

我们将RANSACRegression的最大迭代次数设定为100，设定参数min_samples＝50，即随机抽取作为内点的最小样本数量设定为50。使用residual_metric参数，我们向其传递了一个lambda函数，此函数能够计算拟合曲线与样本点间垂直距离的绝对值。同时将residual_threshold参数的值设定为5.0，这样只有与拟合曲线垂直距离小于5个单位长度的样本点才能加入到内点集合，此设置在特定数据集上表现良好。scikit-learn默认使用MAD估计来设置内点阈值，其中MAD是目标值y的中位数绝对偏差（Median Absolute Deviation）的缩写。不过，内点阈值的确定是与具体问题相关的，这也是RANSAC的一个问题所在。近年来，已经出现了多种能够自动选出适宜的内点阈值的方法。读者可以通过R.Toldo和A.Fusiello的论文了解详细内容[1]。

完成RANSAC模型的拟合后，我们使用该RANSAC线性回归模型来获取内点和异常值的集合，并将它们与使用内点拟合得到的曲线一同绘制：

```
>>> inlier_mask = ransac.inlier_mask_
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...             c='blue', marker='o', label='Inliers')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...             c='lightgreen', marker='s', label='Outliers')
>>> plt.plot(line_X, line_y_ransac, color='red')
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

从下面的散点图中可以看到，线性回归模型是通过内点（圆）集合拟合得到的：



我们使用下面的代码来显示模型的斜率和截距，可以看到结果与前面未使用RANSAC得到的拟合曲线稍有不同：

```
>>> print('Slope: %.3f' % ransac.estimator_.coef_[0])
Slope: 9.621
>>> print('Intercept: %.3f' % ransac.estimator_.intercept_)
Intercept: -37.137
```

使用RANSAC，我们降低了数据集中异常点的潜在影响，但无法确定该方法对未知数据的预测性能是否存在正面影响。因此，下一节中将讨论回归模型评估的不同方法，这是预测模型构建系统中的一个重要组成部分。

[1] Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting,in Image Analysis and Processing-ICIAP 2009,pages 123-131.Springer,2009.

10.5 线性回归模型性能的评估

在前面的小节中，我们讨论了如何在训练数据上拟合回归模型。然而，通过前面章节的学习，我们了解到：为了获得对模型性能的无偏估计，在训练过程中使用未知数据对模型进行测试是至关重要的。

还记得在第6章中，将数据集划分为训练数据集和测试数据集，前者用于模型的拟合，后者用于评估模型在未知数据上的泛化性能。简单回归模型的介绍就到这里，我们现在使用数据集中的所有变量训练多元回归模型：

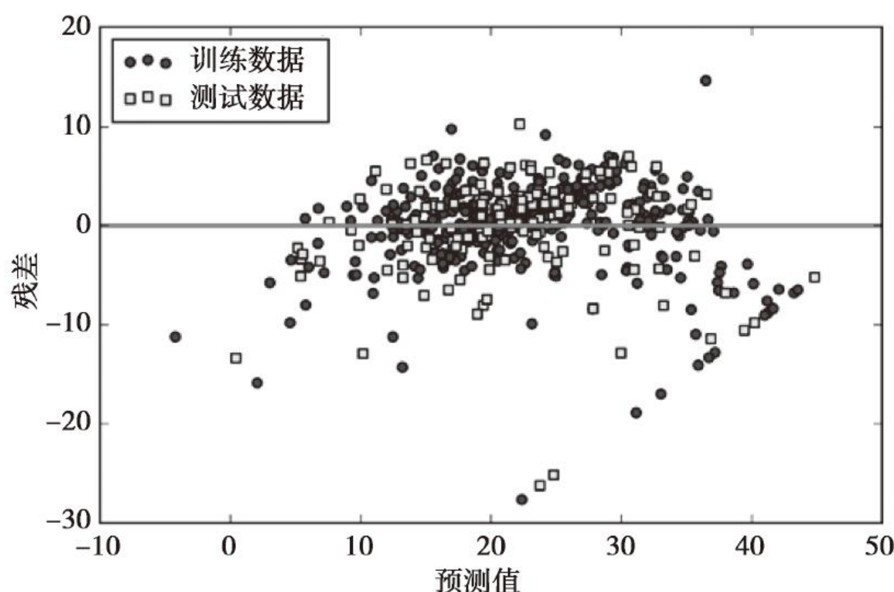
```
>>> from sklearn.cross_validation import train_test_split
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...        X, y, test_size=0.3, random_state=0)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

这个模型使用了多个解释变量，我们无法在二维图上绘制线性回归曲线（更确切地说是超平面），不过可以绘制出预测值的残差（真实值与预测值之间的差异或者垂直距离）图，从而对回归模型进行评估。残差图作为常用的图形分析方法，可对回归模型进行评估、获取模型的异常值，同时还可检查模型是否是线性的，以及误差是否随机分布。

使用如下代码，我们绘制得到残差图，其中，通过将预测结果减去对应目标变量真实值，便可获得残差的值：

```
>>> plt.scatter(y_train_pred, y_train_pred - y_train,
...             c='blue', marker='o', label='Training data')
>>> plt.scatter(y_test_pred,  y_test_pred - y_test,
...             c='lightgreen', marker='s', label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

执行上述代码，我们应该看到如下残差图像，其中包含一条穿过x轴原点的直线，如下图所示：



完美的预测结果其残差应为0，但在实际应用中，这种情况可能永远都不会发生。不过，对于一个好的回归模型，我们期望误差是随机分布的，同时残差也随机分布于中心线附近。如果我们从残差图中找出规律，这意味着模型遗漏了某些能够影响残差的解释信息，就如同刚看到的残差图那样，其中有着些许规律。此外，我们还可以使用残差图来发现异常

值，这些异常值点看上去距离中心线有较大的偏差。

另外一种对模型性能进行定量评估的方法称为均方误差（Mean Squared Error，MSE），它是线性回归模型拟合过程中，最小化误差平方和（SSE）代价函数的平均值。MSE可用于不同回归模型的比较，或是通过网格搜索进行参数调优，以及交叉验证等：

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

执行如下代码：

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
```

可以看到，训练集上的MSE值为19.96，而测试集上的MSE值骤升为27.20，这意味着我们的模型过拟合于训练数据。

某些情况下决定系数（coefficient of determination）（R^2）显得尤为有用，它可以看作是MSE的标准化版本，用于更好地解释模型的性能。换句话说，R^2是模型捕获的响应方差的分数。R^2值的定义如下：

$$R^2 = 1 - \frac{SSE}{SST}$$

其中，SSE 为误差平方和，而 $SST = \sum_{i=1}^{n} (y^{(i)} - \mu_y)^2$，换句话说，$R^2$ 就是预测值的方差。

我们来看一下使用 MSE 定义的 R^2：

$$\begin{aligned} R^2 &= 1 - \frac{SSE}{SST} \\ &= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2} \\ &= 1 - \frac{MSE}{Var(y)} \end{aligned}$$

对于训练数据集来说，R^2的取值范围介于区间[0,1]，对于测试集来说，其值可能为负。如果$R^2 = 1$，此时MSE = 0，这意味着模型完美拟合了数据。

在训练集上进行评估，R^2的值为0.765，看起来还不错。不过，在测试上R^2的值仅为0.673，计算代码如下：

```
>>> from sklearn.metrics import r2_score
>>> print('R^2 train: %.3f, test: %.3f' %
...       (r2_score(y_train, y_train_pred),
...        r2_score(y_test, y_test_pred)))
```

10.6 回归中的正则化方法

正如第3章中所讨论的，正则化是通过在模型中加入额外信息来解决过拟合问题的一种方法，引入罚项增加了模型的复杂性但却降低了模型参数的影响。最常见的正则化线性回归方法就是所谓的岭回归（Ridge Regression）、最小绝对收缩及算子选择（Least Absolute Shrinkage and Selection Operator，LASSO）以及弹性网络（Elastic Net）等。

岭回归是基于L2罚项的模型，我们只是在最小二乘代价函数中加入了权重的平方和：

$$J(w)_{Ridge}=\sum_{i=1}^{n}\left(y^{(i)}-\hat{y}^{(i)}\right)^2+\lambda\|w\|_2^2$$

其中：

$$\text{L2:}\quad \lambda\|w\|_2^2=\lambda\sum_{j=1}^{m}w_j^2$$

通过增加超参的值，我们可以增加正则化的强度，同时也就降低了权重对模型的影响。请注意，正则化项不影响截距项w_0。

对于基于稀疏数据训练的模型，还有另外一种解决方案，即LASSO。基于正则化项的强度，某些权重可以变为零，这也使得LASSO成为一种监督特征选择技术：

$$J(w)_{LASSO}=\sum_{i=1}^{n}\left(y^{(i)}-\hat{y}^{(i)}\right)^2+\lambda\|w\|_1$$

其中：

$$\text{L1:} \quad \lambda\|w\|_1 = \lambda\sum_{j=1}^{m}|w_j|$$

不过LASSO存在一个限制，即如果m > n，则至多可以完成n个变量的筛选。弹性网络则是岭回归和LASSO之间的一个折中，其中包含一个用于稀疏化的L1罚项，以及一个消除LASSO限制（如可筛选变量数量）的L2罚项。

$$J(w)_{ElasticNet} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_1\sum_{j=1}^{m}w_j^2 + \lambda_2\sum_{j=1}^{m}|w_j|$$

scikit-learn中包含前面提及的所有正则化回归模型，除了需要参数指定正则化强度外，这些模型的使用方式与k折交叉验证等普通回归模型的使用方法相同。

岭回归模型的初始化方式如下：

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

请注意，正则化强度通过alpha来调节，它类似于参数。同样，我们可以使用linear_model子模块下的Lasso对象来初始化LASSO回归：

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

最后，scikit-learn下的ElasticNet允许我们调整L1与L2的比率：

```
>>> from sklearn.linear_model import ElasticNet
>>> lasso = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

例如，如果将l1_ratio设置为1.0，此时ElasticNet回归等同于LASSO回归。如想了解线性回归不同实现方式的更多信息请参见：http://scikit-

learn.org/stable/modules/linear_model.html。

10.7 线性回归模型的曲线化-多项式回归

在前面的小节中，我们假定了单一解释变量与响应变量的线性关系。对于不符合线性假设的问题，一种常用的解释方法就是通过加入多项式项来使用多项式回归模型：

$$y = w^0 + w_1 x + w_2 x^2 + \cdots + w_d x^d$$

其中，d为多项式的次数。虽然我们可以使用多项式回归对非线性关系建模，但由于线性回归系数w的缘故，多项式回归仍旧被看作是多元线性回归模型。

我们现在来讨论一下，如何使用scikit-lern中的PolynominalFeatures转换类在只含一个解释变量的简单回归问题中加入二次项（d = 2），并且将多项式回归与线性回归进行线性拟合比较。步骤如下：

1）增加一个二次多项式项：

```
from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([258.0, 270.0, 294.0,
…                            320.0, 342.0, 368.0,
…                            396.0, 446.0, 480.0,
…                            586.0])[:, np.newaxis]

>>> y = np.array([236.4, 234.4, 252.8,
…                           298.6, 314.2, 342.2,
…                           360.8, 368.0, 391.2,
…                           390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

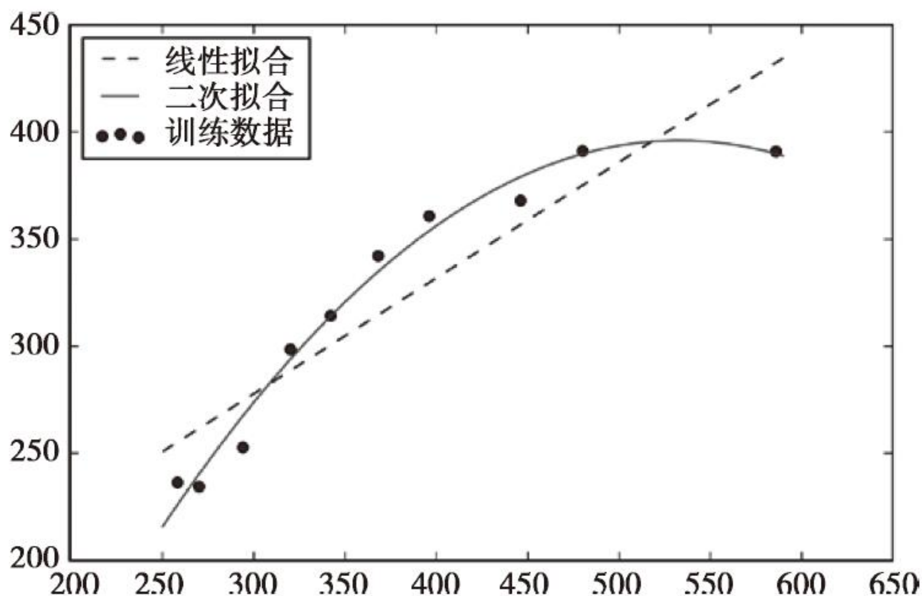
2）拟合一个用于对比的简单线性回归模型：

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250,600,10)[:, np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

3）使用经过转换后的特征针对多项式回归拟合一个多元线性回归模型：

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
Plot the results:
>>> plt.scatter(X, y, label='training points')
>>> plt.plot(X_fit, y_lin_fit,
...          label='linear fit', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...          label='quadratic fit')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

从结果图像中可以看出，与线性拟合相比，多项式拟合可以更好地捕获到解释变量与响应变量之间的关系。



```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> print('Training MSE linear: %.3f, quadratic: %.3f' % (
...         mean_squared_error(y, y_lin_pred),
...         mean_squared_error(y, y_quad_pred)))
Training MSE linear: 569.780, quadratic: 61.330
>>> print('Training  R^2 linear: %.3f, quadratic: %.3f' % (
...         r2_score(y, y_lin_pred),
...         r2_score(y, y_quad_pred)))
Training  R^2 linear: 0.832, quadratic: 0.982
```

执行上述代码后，MSE的值由线性拟合的570下降到了二次拟合的61。同时，与线性拟合的结果（$R^2 = 0.832$）相比，二次模型的判定系数的结果（$R^2 = 0.982$）更好，说明二次拟合在此问题上的效果更佳。

10.7.1　房屋数据集中的非线性关系建模

在上一小节中，我们通过一个简单问题，讨论了如何通过多项式特征来拟合非线性关系。现在来看一个更加具体的例子，并将这些概念应用到房屋数据集中。通过执行下面的代码，我们将使用二次和三次多项式对房屋价格和LSTAT（弱势群体人口所占比例）之间的关系进行建模，并与线性拟合进行对比。

代码如下：

```
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> regr = LinearRegression()

# create polynomial features
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)

# linear fit
>>> X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]
>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))

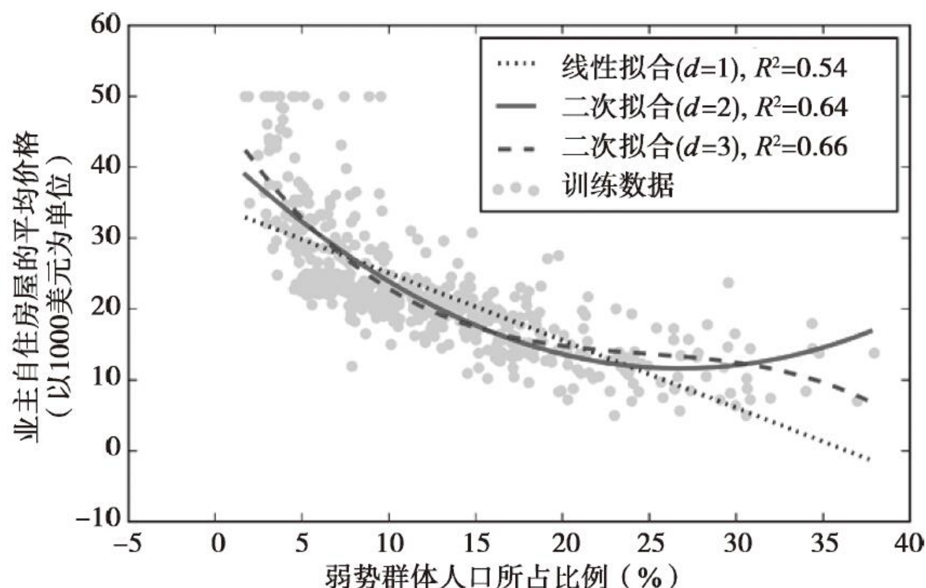
# quadratic fit
>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))

# cubic fit
>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))

# plot results
>>> plt.scatter(X, y,
...             label='training points',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
```

```
...          label='linear (d=1), $R^2=%.2f$'
...            % linear_r2,
...          color='blue',
...          lw=2,
...          linestyle=':')
>>> plt.plot(X_fit, y_quad_fit,
...          label='quadratic (d=2), $R^2=%.2f$'
...            % quadratic_r2,
...          color='red',
...          lw=2,
...          linestyle='-')
>>> plt.plot(X_fit, y_cubic_fit,
...          label='cubic (d=3), $R^2=%.2f$'
...            % cubic_r2,
...          color='green',
...          lw=2,
...          linestyle='--')
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.legend(loc='upper right')
>>> plt.show()
```

由结果图像可知，相较于线性拟合和二次拟合，三次拟合更好地捕获了房屋价格与LSTAT之间的关系。不过，我们应该意识到，加入越来越多的多项式特征会增加模型的复杂度，从而更易导致过拟合。由此，在实际应用中，建议在单独的测试数据集上评价模型的性能，进而对其泛化性能进行评估：



此外，多项式特征并非总是非线性关系建模的最佳选择。例如，仅就MEDV-LSTAT的散点图来说，我们可将LSTAT特征变量的对数值以及MEDV的平方根映射到一个线性特征空间，并使用线性回归进行拟合。可通过执行下面的代码对此假设进行验证：

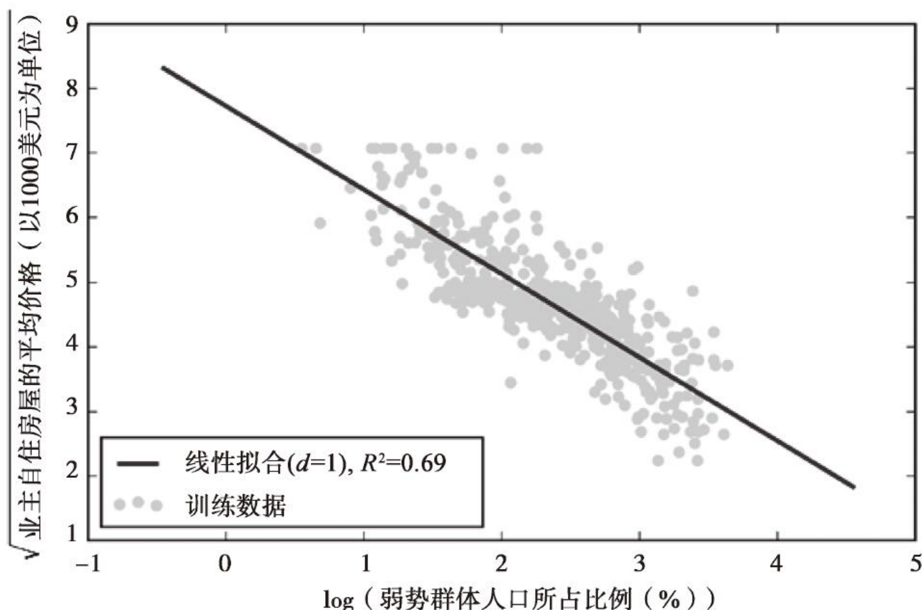
```
# transform features
>>> X_log = np.log(X)
>>> y_sqrt = np.sqrt(y)

# fit features
>>> X_fit = np.arange(X_log.min()-1,
...                    X_log.max()+1, 1)[:, np.newaxis]
```

```
>>> regr = regr.fit(X_log, y_sqrt)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y_sqrt, regr.predict(X_log))

# plot results
>>> plt.scatter(X_log, y_sqrt,
...             label='training points',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label='linear (d=1), $R^2=%.2f$' % linear_r2,
...          color='blue',
...          lw=2)
>>> plt.xlabel('log(% lower status of the population [LSTAT])')
>>> plt.ylabel('$\sqrt{Price \; in \; \$1000\'s [MEDV]}$')
>>> plt.legend(loc='lower left')
>>> plt.show()
```

在将解释变量映射到对数空间以及取目标变量的平方根后，我们可以捕获到两者之间的线性关系，其拟合度$R^2 = 0.69$看似优于前面使用的任何一种多项式回归：



10.7.2　使用随机森林处理非线性关系

在本节，我们将了解一下随机森林（random forest）回归，它从概念上异于本章中介绍的其他回归模型。随机森林是多棵决策树（decision tree）的集合，与先前介绍的线性回归和多项式回归不同，它可以被理解为分段线性函数的集成。换句话说，通过决策树算法，我们把输入空间细分为更小的区域以更好地管理。

1.决策树回归

决策树算法的一个优点在于：如果我们处理的是非线性数据，无需对其进行特征转换。在第3章中曾经介绍过，我们迭代地对节点进行划分来构建决策树，直到所有叶子节点的不纯度为0，或者满足终止条件时才停止迭代。当使用决策树进行分类时，定义熵作为不纯度的衡量标准，旨在通过最大信息增益（Information Gain，IG）对特征进行划分，由此可定义如下二分规则：

$$IG(D_p, x) = I(D_p) - \frac{1}{N_p} I$$

其中，x为待划分特征，N_p为父节点中样本数量，I为不纯度函数，D_p为父节点中训练样本的子集，（D）和（D）为划分后所有节点中样本的数量。请记住，我们的目标是通过特征的划分来使得信息增益最大化，换句话说，我们试图找到使得子节点中信息不纯度最低的特征划分。在第3章中，我们使用熵作为不纯度度量，这是用于分类的一个有效标准。为了将决策树用于回归，我们使用MSE替代熵作为节点t的不纯度度量标准：

$$I(t)=MSE(t)=\frac{1}{N_t}\sum_{i\in D_t}\left(y^{(i)}-\hat{y}_t\right)^2$$

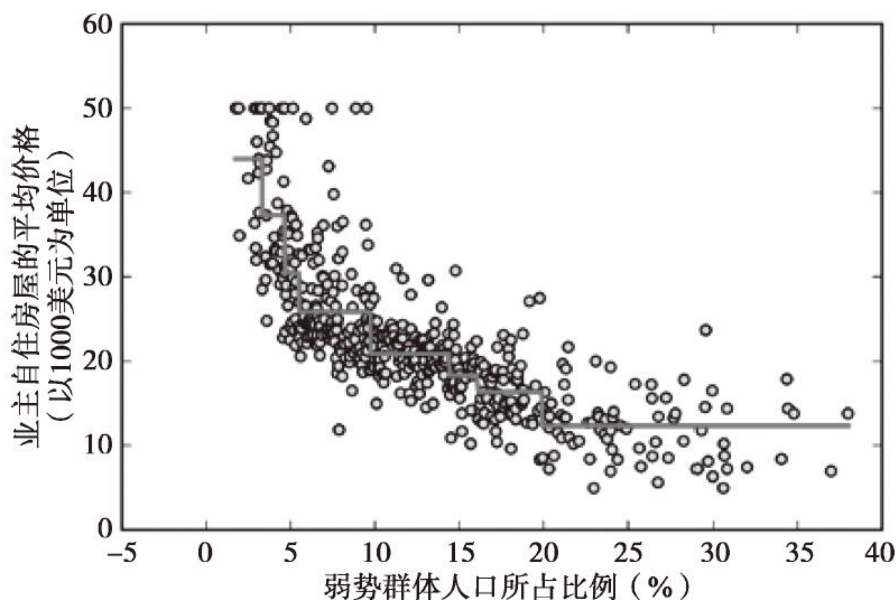
其中，N_t为节点t中训练样本的数量，D_t为节点t中训练样本的子集，$y^{(i)}$为真实的目标值，$\hat{y}_t$为预测目标值（样本均值）：

$$\hat{y}_t=\frac{1}{N}\sum_{i\in D_t}y^{(i)}$$

MSE用于决策树回归时，也常称为节点内方差，这就是分裂标准常称为方差缩减（variance reduction）的原因。为了知道决策树的拟合效果，我们使用scikit-learn中的DecisionTreeRegressor类对MEDV和LSTAT两个变量之间的非线性关系进行建模：

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
>>> tree.fit(X, y)
>>> sort_idx = X.flatten().argsort()
>>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000\'s [MEDV]')
>>> plt.show()
```

从结果图中可以看到，决策树捕捉到了数据的整体趋势。不过，此模型的一个局限在于：它无法捕获期望预测的连续性与可导性。此外，我们还需注意要为树选择合适的深度，以免造成过拟合或者欠拟合，在此例中，深度为3的树看起来是比较合适的：



下一节，我们将学习一种更鲁棒的决策树拟合方法：随机森林。

2.随机森林回归

正如我们在第3章中讨论的那样，随机森林算法是组合多棵决策树的一种集成技术。由于随机性有助于降低模型的方差，与单棵决策树相比，随机森林通常具有更好的泛化性能。随机森林的另一个优势在于：它对数据集中的异常值不敏感，且无需过多的参数调优。随机森林中唯一需要我们通过实验来获得的参数就是待集成决策树的数量。随机森林用于回归的基本方法与我们在第3章中讨论过的随机森林用于分类的方法基本一致。唯一区别在于：随机森林回归使用MSE作为单棵决策树生成的标准，同时所有决策树预测值的平均数作为预测目标变量的值。

现在，我们使用房屋数据集中的所有特征来拟合一个随机森林回归模

型，其中60%的样本用于模型的训练，剩余的40%用来对模型进行评估，代码如下：

```
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test =\
...       train_test_split(X, y,
...                        test_size=0.4,
...                        random_state=1)

>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(                              ..
.                                n_estimators=1000,
...                                criterion='mse',
...                                random_state=1,
...                                n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> y_train_pred = forest.predict(X_train)
>>> y_test_pred = forest.predict(X_test)
>>> print('MSE train: %.3f, test: %.3f' % (
...        mean_squared_error(y_train, y_train_pred),
...        mean_squared_error(y_test, y_test_pred)))
>>> print('R^2 train: %.3f, test: %.3f' % (
...        r2_score(y_train, y_train_pred),
...        r2_score(y_test, y_test_pred)))
MSE train: 3.235, test: 11.635
R^2 train: 0.960, test: 0.871
```

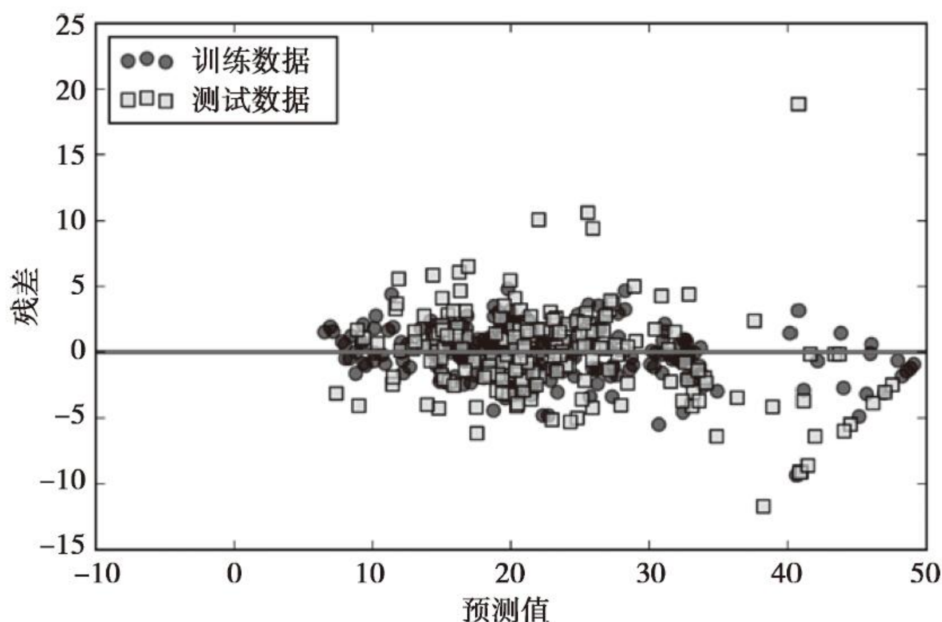
遗憾的是，我们发现随机森林对于训练数据有些过拟合。不过，它仍旧能够较好地解释目标变量与解释变量之间的关系（在测试数据集上，$R^2 = 0.871$）。

最后，让我们来看一下预测的残差：

```
>>> plt.scatter(y_train_pred,
...             y_train_pred - y_train,
...             c='black',
...             marker='o',
...             s=35,
...             alpha=0.5,
...             label='Training data')
>>> plt.scatter(y_test_pred,
...             y_test_pred - y_test,
...             c='lightgreen',
...             marker='s',
...             s=35,
...             alpha=0.7,
...             label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')

>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
>>> plt.xlim([-10, 50])
>>> plt.show()
```

根据决定系数R^2的值可知，模型在训练数据上的拟合效果要好于测试数据，这与下图中y轴方向出现异常值所反应的情况一致。此外，残差没有完全随机分布在中心点附近，这意味着模型无法捕获所有的解释信息。不过，与本章前面小节中绘制的线性模型的残差图相比，随机森林回归的残差图有了很大的改进：



在第3章中，我们已经讨论过核技巧，并将其应用到了支持向量机（SVM）的分类中，若需要处理非线性问题，此技巧是非常有用的。虽然此话题已经超出了本书的范围，但是支持向量机的确可以用于非线性回归任务。关于支持向量机在回归中的应用问题，S.R.Gunn在其报告中做了精彩的论述：S.R.Gunn et al.Vector Machines for Classification and Regression.（ISIS technical report,14,1998），感兴趣的读者可以通过此文献了解更多信息。scikit-learn中已经实现了支持向量机的回归模型，关于其详细信息请见链接：http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR。

本章小结

本章开始，我们学习了如何使用简单回归模型对单个解释变量和连续目标变量之间的关系进行建模。进而，我们讨论了一种用于了解数据中模式与异常点的解释性数据分析技术，这是预测模型构建中重要的一步。

基于梯度优化的方法，我们构建并实现了第1个线性回归模型。我们学习了如何使用scikit-learn中的线性回归模型，并且针对异常值的处理实现了一个鲁棒的线性回归模型（RANSAC）。为了更深入了解回归模型的预测性能，我们计算了误差平方和的平均值，以及R2等衡量标准。此外，我们还讨论了一种诊断回归模型中所存在问题的图像化方法：残差图。

之后，我们讨论了如何将正则化方法应用于回归模型，以降低模型复杂度及避免过拟合，此外还介绍了非线性关系建模的几种方法，包括多项式特征转换和随机森林回归。

在前面的章节中，我们详细介绍了监督学习、分类以及回归分析的相关内容。下一章中，我们将讨论机器学习的另一个有趣的子领域：无监督学习。届时，读者将学到如何使用聚类分析在无目标变量的情况下挖掘数据中的潜在结构。

第11章　聚类分析——处理无类标数据

在前面的章节中，使用了监督学习技术来构建机器学习模型，其中训练数据都是事先已知预测结果的，即训练数据中已提供了数据的类标。在本章中，我们将转而研究聚类分析，它是一种无监督学习（unsupervised learning）技术，可以在事先不知道正确结果（即无类标信息或预期输出值）的情况下，发现数据本身所蕴含的结构等信息。聚类的目标是发现数据中自然形成的分组，使得每个簇内样本的相似性大于与其他簇内样本的相似性。

由于聚类本身带有探索的性质，因此更能激发人们的兴趣。本章通过学习如下概念，我们能够更加有效地组织数据：

·使用k-means算法发现簇中心

·使用自底向上的方法构建层次聚类树

·基于密度聚类方法发现任意形状簇

11.1　使用k-means算法对相似对象进行分组

本节将讨论一种最流行的聚类算法：k-means算法，它在学术领域及业界都得到了广泛应用。聚类（或称为聚类分析）是一种可以找到相似对象群组的技术，与组间对象相比，组内对象之间具有更高的相似度。聚类在商业领域的应用包括：按照不同主题对文档、音乐、电影等进行分组，或基于常见的购买行为，发现有相同兴趣爱好的顾客，并以此构建推荐引擎。

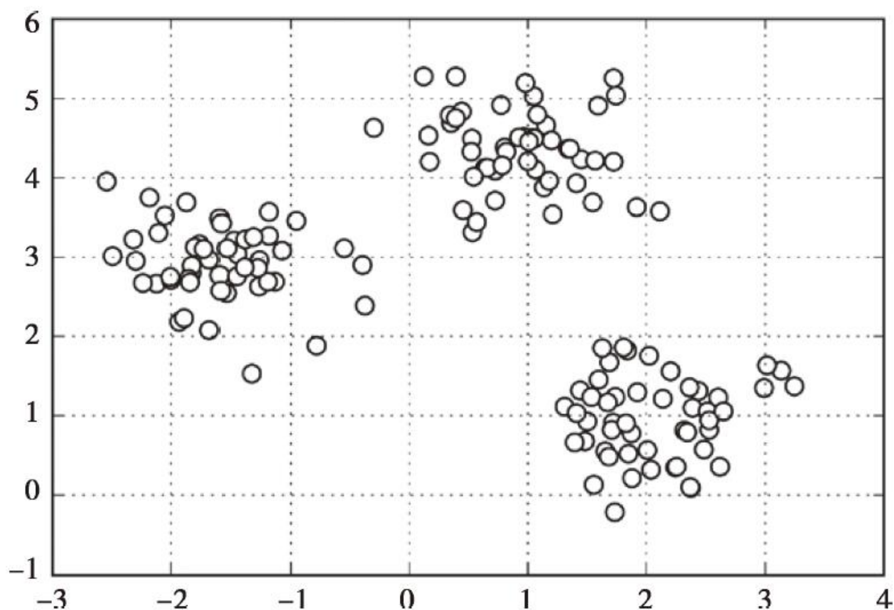
读者稍后将看到，与其他聚类算法相比，k-means算法易于实现，且具有很高的计算效率，这也许就是它得到广泛使用的原因。k-means算法是基于原型的聚类。在本章的后续内容中，我们还将讨论另外两种聚类：层次（hierarchical）聚类和基于密度（density-based）的聚类。基于原型的聚类意味着每个簇都对应一个原型，它可以是一些具有连续型特征的相似点的中心点（centroid）（平均值），或者是类别特征情况下相似点的众数（medoid）——最典型或是出现频率最高的点。虽然k-means算法可以高效识别球形簇，但是此算法的缺点在于必须事先指定先验的簇数量k。如果k值选择不当，则可能导致聚类效果不佳。本章后续还将讨论肘（elbow）方法和轮廓图（silhouette plot），它们可以用来评估聚类效果，并帮助我们选出最优k值。

尽管k-means聚类适用于高维数据，但出于可视化需要，我们将使用一个二维数据集的例子进行演示：

```
>>> from sklearn.datasets import make_blobs
>>> X, y = make_blobs(n_samples=150,
...                   n_features=2,
...                   centers=3,
...                   cluster_std=0.5,
...                   shuffle=True,
...                   random_state=0)

>>> import matplotlib.pyplot as plt
>>> plt.scatter(X[:,0],
...             X[:,1],
...             c='white',
...             marker='o',
...             s=50)
>>> plt.grid()
>>> plt.show()
```

我们刚才创建的数据集中包含150个随机生成的点，它们大致分为三个高密度区域，其二维散点图如下：



在聚类算法的实际应用中，我们没有任何关于这些样本的类别基础信

息；否则算法就要划分到监督学习的范畴了。由此，我们的目标就是根据样本自身特征的相似性对其进行分组，对此可采用k-means算法，具体有如下四个步骤：

1）从样本点中随机选择k个点作为初始簇中心。

2）将每个样本点划分到距离它最近的中心点$\mu^{(j)}, j\in\{1,\ldots,k\}$所代表的簇中。

3）用各簇中所有样本的中心点替代原有的中心点。

4）重复步骤2和3，直到中心点不变或者达到预定迭代次数时，算法终止。

现在面临的一个问题就是：如何度量对象之间的相似性？我们可以将相似性定义为距离的倒数，在m维空间中，对于特征取值为连续型实数的聚类分析来说，常用的距离度量标准是欧几里得距离的平方：

$$d(\boldsymbol{x}, \boldsymbol{y})^2=\sum_{j=1}^{m}(x_j-y_j)^2=\|\boldsymbol{x}-\boldsymbol{y}\|_2^2$$

请注意，在前面的公式中，下标索引j为样本点x和y的第j个维度（特征列）。本节后续内容将分别使用上标i和j来代表样本索引和簇索引。

基于欧几里得度量标准，我们可以将k-means算法描述为一个简单的优化问题，通过迭代使得簇内误差平方和（within-cluster sum of squared errors，SSE）最小，也称作簇惯性（cluster intertia）。

$$SSE=\sum_{i=1}^{n}\sum_{j=1}^{k}w^{(i,j)}=\left\|\boldsymbol{x}^{(i)}-\boldsymbol{\mu}^{(j)}\right\|_2^2$$

其中，$\mu^{(j)}$为簇j的中心点，如果样本$x^{(i)}$属于簇j，则有$w^{(i,j)}=1$，否则$w^{(i,j)}=0$。

至此，我们已经知道了简单k-means算法的工作原理，现在借助scikit-learn中的KMeans类将k-means算法应用于我们的示例数据集：

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
...             init='random',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
```

使用上述代码，我们将簇数量设定为3个；指定先验的簇数量是k-means算法的一个缺陷，设置n_init = 10，程序能够基于不同的随机初始中心点独立运行算法10次，并从中选择SSE最小的作为最终模型。通过max_iter参数，指定算法每轮运行的迭代次数（在本例中为300次）。请注意，在scikit-learn对k-means算法的实现中，如果模型收敛了，即使未达到预定迭代次数，算法也将终止。

不过，在k-means算法的某轮迭代中，可能会发生无法收敛的情况，特别是当我们设置了较大的max_iter值时，更有可能产生此类问题。解决收敛问题的一个方法就是为tol参数设置一个较大的值，此参数控制对簇内误差平方和的容忍度，此容忍度用于判定算法是否收敛。在上述代码中，我们设置的容忍度为1e-04（0.0001）。

11.1.1　k-means++

到目前为止，我们已经讨论了经典的k-means算法，它使用随机点作为初始中心点，若初始中心点选择不当，有可能会导致簇效果不佳或产生收敛速度慢等问题。解决此问题的一种方案就是在数据集上多次运行k-means算法，并根据误差平方和（SSE）选择性能最好的模型。另一种方案就是使用k-means++算法让初始中心点彼此尽可能远离，相比传统k-means算法，它能够产生更好、更一致的结果。[1]

k-means++算法的初始化过程可以概括如下：

1）初始化一个空的集合M，用于存储选定的k个中心点。

2）从输入样本中随机选定第一个中心点$\mu^{(j)}$，并将其加入到集合M中。

3）对于集合M之外的任一样本点$x^{(i)}$，通过计算找到与其平方距离最小的样本$d(x^{(i)},M)^2$。

4）使用加权概率分布$\frac{d(\boldsymbol{\mu}^{(p)}, \mathbf{M})^2}{\sum_i d(\boldsymbol{x}^{(i)}, \mathbf{M})^2}$来随机选择下一个中心点$\mu^{(p)}$。

5）重复步骤2、3，直到选定k个中心点。

6）基于选定的中心点执行k-means算法。

通过scikit-learn的KMeans对象来实现k-means++算法，只需将init参数的值random替换为k-means++（默认值）即可。

k-means算法还有另一个问题，就是一个或多个簇的结果可能为空。但k-medoids或者模糊C-means算法中不存在这种问题，我们将在下一小节讨论这两种算法。不过，此问题在当前scikit-learn实现的k-means算法中是存在的。如果某个簇为空，算法将搜索距离空簇中心点最远的样本，然后将此最远样本点作为中心点。

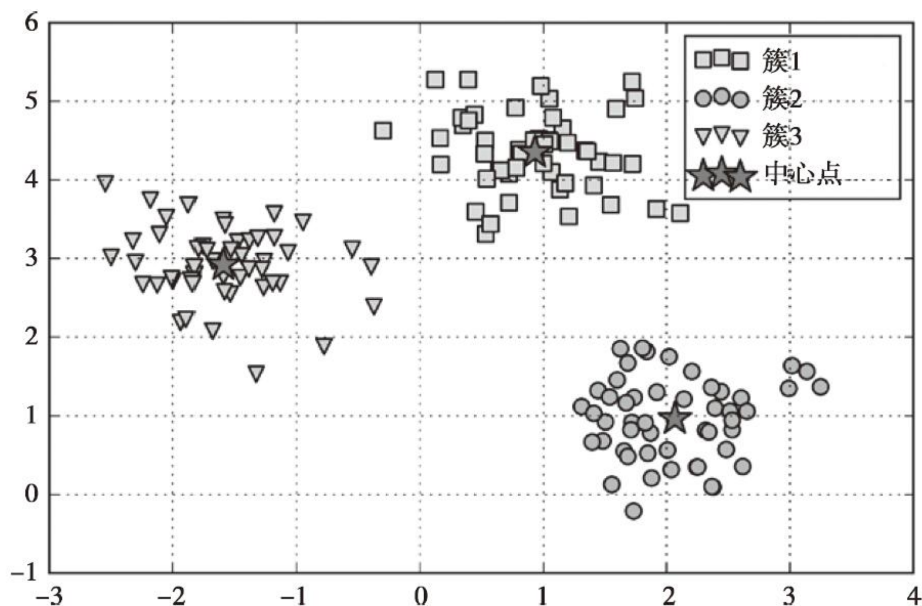
当我们使用欧几里得距离作为度量标准将k-means算法应用到真实数据中时，需要保证所有特征值的范围处于相同尺度，必要时可使用z-score标准化或最小-最大缩放方法对数据进行预处理。

我们已经将簇结果存储在y_km中，并且讨论了k-means算法面临的挑战，现在对k-means算法的簇结果及其相应的簇中心做可视化展示。簇中心保存在KMeans对象的centers_属性中：

```
>>> plt.scatter(X[y_km==0,0],
...             X[y_km ==0,1],
...             s=50,
...             c='lightgreen',
...             marker='s',
...             label='cluster 1')
>>> plt.scatter(X[y_km ==1,0],
...             X[y_km ==1,1],
...             s=50,
...             c='orange',
...             marker='o',
...             label='cluster 2')
>>> plt.scatter(X[y_km ==2,0],
...             X[y_km ==2,1],
...             s=50,
...             c='lightblue',
...             marker='v',
...             label='cluster 3')
>>> plt.scatter(km.cluster_centers_[:,0],
...             km.cluster_centers_[:,1],
...             s=250,
...             marker='*',
...             c='red',
...             label='centroids')
>>> plt.legend()
>>> plt.grid()
>>> plt.show()
```

在下面的散点图中，我们可以看到，通过k-means算法得到的3个中心点位于各球状簇的圆心位置，在此数据集上，分组结果看起来是合理的。

虽然k-means算法在这一简单数据集上运行良好，但我们还要注意k-means算法面临的一些挑战。k-means的缺点之一就是我们必须指定一个先验的簇数量k，但在实际应用中，簇数量并非总是显而易见的，特别当我们面对的是一个无法可视化展现的高维数据集。k-means的另一个特点就是簇不可重叠，也不可分层，并且假定每个簇至少会有一个样本。



[1] D. Arthur and S. Vassilvitskii. k-means++: The Advantages of Careful Seeding. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, pages 1027-1035. Society for Industrial and Applied Mathematics, 2007.

11.1.2　硬聚类与软聚类

硬聚类（hard clustering）指的是数据集中每个样本只能划至一个簇的算法，例如我们在上一小节中讨论过的k-means算法。相反，软聚类（soft clustering，有时也称作模糊聚类（fuzzy clustering））算法可以将一个样本划分到一个或者多个簇。一个常见的软聚类算法是模糊C-means（fuzzy C-means，FCM）算法（也称为soft k-means或者fuzzy k-means）。关于软聚类的最初构想可以追溯到20世纪70年代，当时Joseph C.Dunn第一个提出了模糊聚类的早期版本，以提高k-means的性能[1]。10年之后，James C.Bedzek发表了他对模糊聚类算法进行改进的研究成果，即FCM算法[2]。

FCM的处理过程与k-means十分相似。但是，我们使用每个样本点隶属于各簇的概率来替代硬聚类的划分。在k-means中，我们使用二进制稀疏向量来表示各簇所含样本：

$$\begin{bmatrix} \mu^{(1)} \to 0 \\ \mu^{(2)} \to 1 \\ \mu^{(3)} \to 0 \end{bmatrix}$$

其中，位置索引的值为1表示样本属于簇中心μ（j）所在的簇（假定k＝3，则j∈{1,2,3}）。相反，FMC中的成员隶属向量可以表示如下：

$$\begin{bmatrix} \mu^{(1)} \to 0.1 \\ \mu^{(2)} \to 0.85 \\ \mu^{(3)} \to 0.05 \end{bmatrix}$$

这里，每个值都在区间[0,1]内，代表样本属于相应簇中心所在簇的概

率。对于给定样本，其隶属于各簇的概率之和为1。与k-means算法类似，我们可以将FCM算法总结为四个核心步骤：

1）指定k个中心点，并随机将每个样本点划分至某个簇。

2）计算各簇中心$\mu^{(j)}$,j∈{1,…,k}。

3）更新各样本点所属簇的成员隶属度。

4）重复步骤2、3，直到各样本点所属簇成员隶属度不变，或是达到用户自定义的容差阈值或最大迭代次数。

可以将FCM的目标函数简写为J_m，它看起来与k-means中需要进行最小化计算的簇内误差平方和（within cluster sum-squared-error）很相似：

$$J_m=\sum_{i=1}^{n}\sum_{j=1}^{k}w^{m(i,j)}\left\|\boldsymbol{x}^{(i)}-\boldsymbol{\mu}^{(j)}\right\|_2^2,\quad m\in[1,\infty]$$

不过请注意：此处的成员隶属度$w^{(i,j)}$不同于k-means算法中的二进制值（$w^{(i,j)}\in\{0,1\}$），它是一个实数，表示样本隶属于某个簇的概率（$w^{(i,j)}\in[0,1]$）。读者也许注意到了，我们在$w^{(i,j)}$中增加了额外的一个指数m，一般情况下其取值范围是大于等于1的整数（通常m＝2），也称为模糊系数（fuzziness coefficient，或直接就叫模糊器（fuzzzifier）），用来控制模糊的程度。模糊聚类中，m的值越大则成员隶属度$w^{(i,j)}$越小。样本点属于某个簇的概率计算公式如下：

$$w^{(i,j)}=\left[\sum_{p=1}^{k}\left(\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{\mu}^{(j)}\right\|_2}{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{\mu}^{(p)}\right\|_2}\right)^{\frac{2}{m-1}}\right]^{-1}$$

如同前面的k-means例子，在此依旧使用3个簇中心，可通过以下方式计算出样本x（i）属于簇μ（j）的隶属度：

$$w^{(i,j)}=\left[\left(\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{\mu}^{(j)}\right\|_2}{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{\mu}^{(1)}\right\|_2}\right)^{\frac{2}{m-1}}+\left(\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{\mu}^{(j)}\right\|_2}{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{\mu}^{(2)}\right\|_2}\right)^{\frac{2}{m-1}}+\left(\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{\mu}^{(j)}\right\|_2}{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{\mu}^{(3)}\right\|_2}\right)^{\frac{2}{m-1}}\right]^{-1}$$

以样本属于特定簇的隶属度为权重，此簇中心μ（j）可以通过所有样本的加权均值计算得到：

$$\boldsymbol{\mu}^{(j)}=\frac{\sum_{i=1}^{n} w^{m(i,j)}\boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} w^{m(i,j)}}$$

仅就计算样本数据簇的隶属度公式来说，直观上看，FCM的单次迭代计算成本比k-means的要高，但是FCM通常只需较少的迭代次数便能收敛。遗憾的是，scikit-learn目前还未实现FCM算法。不过，正如Soumi Ghosh与Sanjay K.Dubey的研究[3]所述，实际应用中k-means和FCM算法通常会得到较为相似的结果。

[1] J. C. Dunn. A Fuzzy Relative of the Isodata Process and its Use in Detecting Compact Well-separated Clusters. 1973.

[2] J. C. Bezdek. Pattern Recognition with Fuzzy Objective Function Algorithms. Springer Science & Business Media, 2013.

[3] S. Ghosh and S. K. Dubey. Comparative Analysis of k-means and Fuzzy c-means Algorithms. IJACSA, 4:35-38, 2013.

11.1.3　使用肘方法确定簇的最佳数量

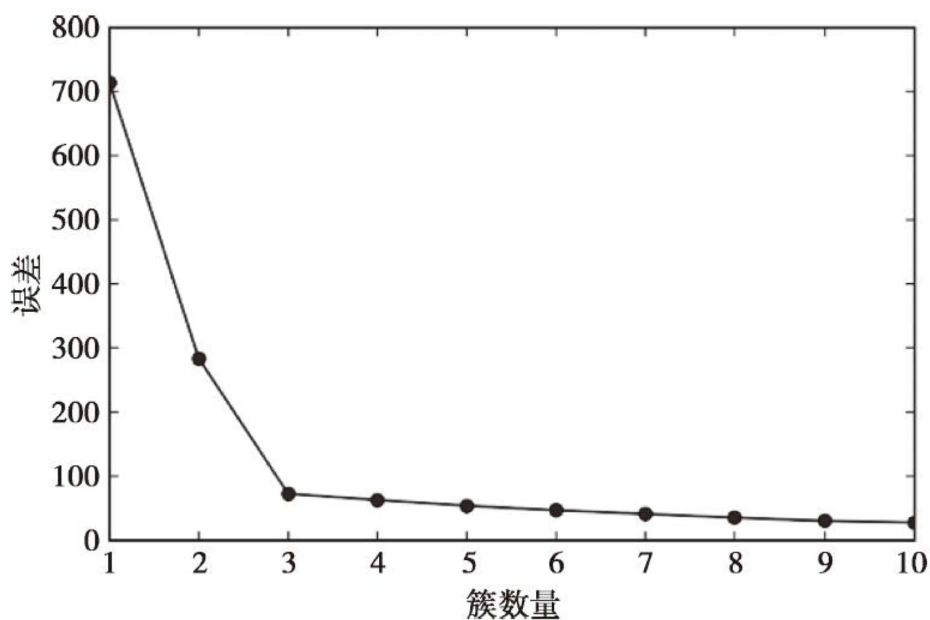
无监督学习中存在一个问题，就是我们并不知道问题的确切答案。由于没有数据集样本类标的确切数据，所以我们无法在无监督学习中使用第6章中用来评估监督学习模型性能的相关技术。因此，为了对聚类效果进行定量分析，我们需要使用模型内部的固有度量来比较不同k-means聚类结果的性能，例如本章先前讨论过的簇内误差平方和（即聚类偏差）。在完成KMeans模型的拟合后，簇内误差平方和可以通过inertia属性来访问，因此，我们无需再次计算就可直接拿来使用。

```
>>> print('Distortion: %.2f' % km.inertia_)
Distortion: 72.48
```

基于簇内误差平方和，我们可使用图形工具，即所谓的肘方法，针对给定任务估计出最优的簇数量k。直观地看，增加k的值可以降低聚类偏差。这是因为样本会更加接近其所在簇的中心点。肘方法的基本理念就是找出聚类偏差骤增时的k值，我们可以绘制出不同k值对应的聚类偏差图，以做更清晰的观察：

```
>>> distortions = []
>>> for i in range(1, 11):
...     km = KMeans(n_clusters=i,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 random_state=0)
>>>     km.fit(X)
>>>     distortions.append(km.inertia_)
>>> plt.plot(range(1,11), distortions, marker='o')
>>> plt.xlabel('Number of clusters')
>>> plt.ylabel('Distortion')
>>> plt.show()
```

如下图所示，当k = 3时图案呈现了肘型，这表明对于此数据集来说，k = 3的确是一个好的选择：



11.1.4 通过轮廓图定量分析聚类质量

另一种评估聚类质量的定量分析方法是轮廓分析（silhouette analysis），此方法也可用于k-means之外的其他聚类方法。轮廓分析可以使用一个图形工具来度量簇中样本聚集的密集程度。通过如下三个步骤，我们可以计算数据集中单个样本的轮廓系数（silhouette coefficient）：

1）将某一样本$x^{(i)}$与簇内其他点之间的平均距离看作是簇的内聚度$a^{(i)}$。

2）将样本$x^{(i)}$与其最近簇中所有点之间的平均距离看作是与下一最近簇的分离度$b^{(i)}$。

3）将簇分离度与簇内聚度之差除以二者中的较大者得到轮廓系数，如下式所示：

$$s^{(i)}=\frac{b^{(i)}-a^{(i)}}{\max\{b^{(i)}, a^{(i)}\}}$$

轮廓系数的值介于-1到1之间。从上式可见，若簇内聚度与分离度相等（$b^{(i)}=a^{(i)}$），则轮廓系数值为0。此外，由于$b^{(i)}$衡量样本与其他簇内样本间的差异程度，而$a^{(i)}$表示样本与簇内其他样本的相似程度，因此，如果$b^{(i)}>>a^{(i)}$，我们可以近似得到一个值为1的理想的轮廓系数。

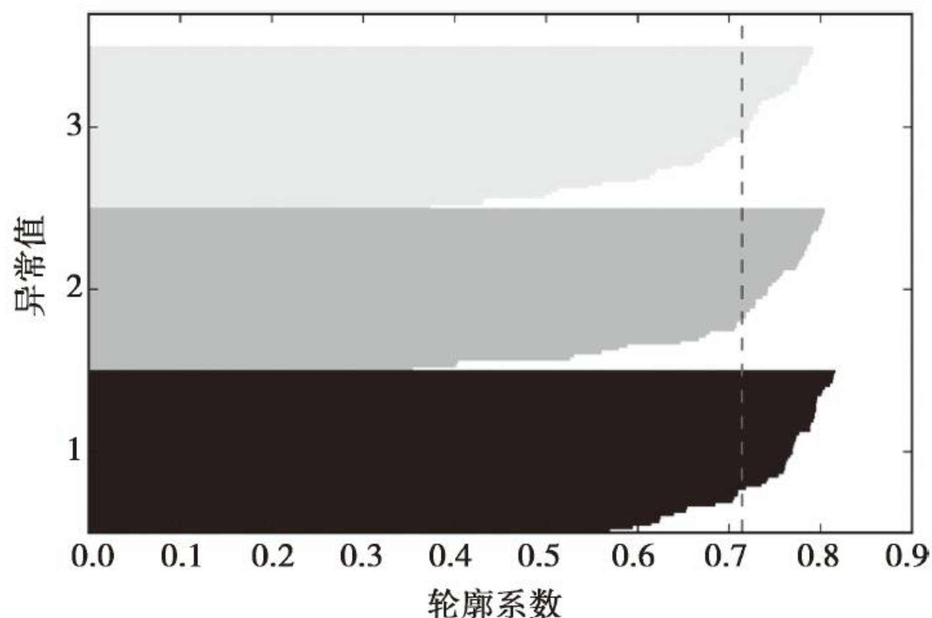
轮廓系数可通过scikit-learn中metric模块下的silhouette_samples计算得到，也可选择使用silhouette_scores。后者计算所有样本点的轮廓系数

均值，等价于numpy.mean（silhouette_samples（…））。执行下面的代码，我们可以绘制出k = 3时k-means算法的轮廓系数图：

```
>>> km = KMeans(n_clusters=3,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)

>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                      y_km,
...                                      metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(i / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...             color="red",
...             linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.show()
```

通过观察轮廓图，我们可以快速知晓不同簇的大小，而且能够判断出簇中是否包含异常点。



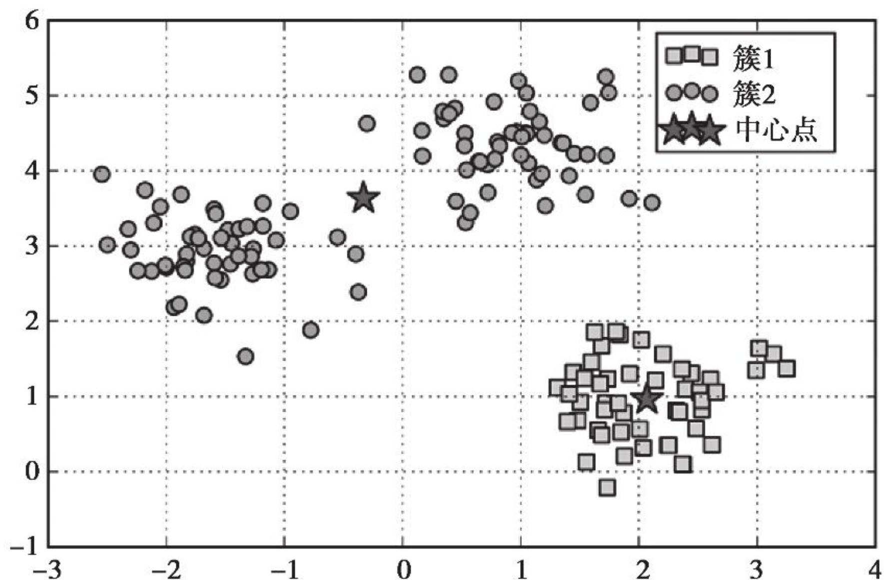
由上图可见，轮廓系数未接近0点，此指标显示聚类效果不错。此外，为了评判聚类效果的优劣，我们在图中增加了轮廓系数的平均值（虚线）。

为了解聚类效果不佳的轮廓图的形状，我们使用两个中心点来初始化k-means算法：

```
>>> km = KMeans(n_clusters=2,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)

>>> plt.scatter(X[y_km==0,0],
...             X[y_km==0,1],
...             s=50, c='lightgreen',
...             marker='s',
...             label='cluster 1')
>>> plt.scatter(X[y_km==1,0],
...             X[y_km==1,1],
...             s=50,
...             c='orange',
...             marker='o',
...             label='cluster 2')
>>> plt.scatter(km.cluster_centers_[:,0],
...             km.cluster_centers_[:,1],
...             s=250,
...             marker='*',
...             c='red',
...             label='centroids')
>>> plt.legend()
>>> plt.grid()
>>> plt.show()
```

由下图可见，样本点形成了三个球状分组，其中一个中心点落在了两个球状分组的中间。尽管聚类结果看上去不是特别糟糕，但它并不是最合适的结果。

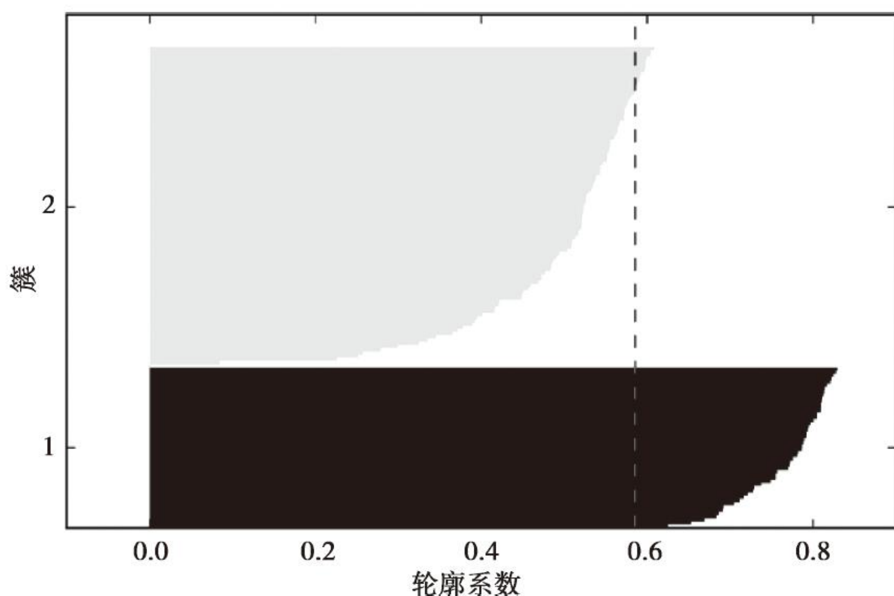


接下来，我们绘制轮库图对聚类结果进行评估。请记住，在实际应用中，我们常常处理的是高维数据，因此并不奢望使用二维散点图对数据集进行可视化。

```
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
```

```
...                                          y_km,
...                                          metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(i / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.show()
```

由结果图像可见，轮廓具有明显不同的长度和宽度，这更进一步证明该聚类并非最优结果。

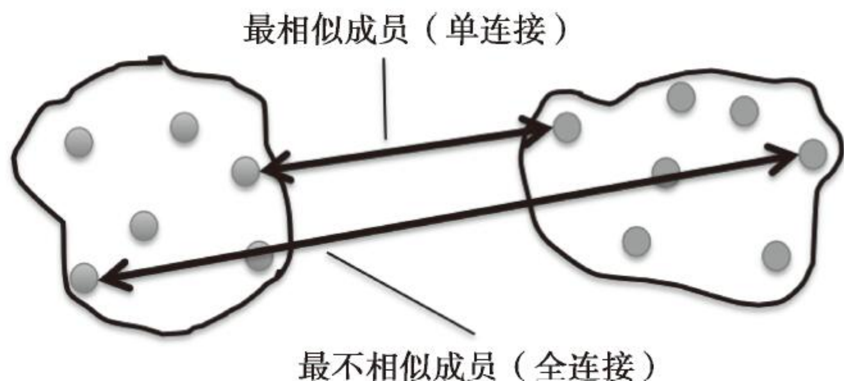


11.2　层次聚类

在本节，我们将学习另一种基于原型的聚类：层次聚类（hierarchical clustering）。层次聚类算法的一个优势在于：它能够使我们绘制出树状图（dendrogram，基于二叉层次聚类的可视化），这有助于我们使用有意义的分类法解释聚类结果。层次聚类的另一优势在于我们无需事先指定簇数量。

层次聚类有两种主要方法：凝聚（agglomerative）层次聚类和分裂（divisive）层次聚类。在分裂层次聚类中，我们首先把所有样本看作是在同一个簇中，然后迭代地将簇划分为更小的簇，直到每个簇只包含一个样本。本节我们将主要介绍凝聚层次聚类，它与分裂层次聚类相反，最初我们把每个样本都看作是一个单独的簇，重复地将最近的一对簇进行合并，直到所有的样本都在一个簇中为止。

在凝聚层次聚类中，判定簇间距离的两个标准方法分别是单连接（single linkage）和全连接（complete linkage）。我们可以使用单连接方法计算每一对簇中最相似两个样本的距离，并合并距离最近的两个样本所属簇。与之相反，全连接的方法是通过比较找到分布于两个簇中最不相似的样本（距离最远的样本），进而完成簇的合并。如下图所示：



凝聚层次聚类中其他常用的算法还有平均连接（average linkage）和ward连接（Ward沮linkage）。使用平均连接时，合并两个簇间所有成员间平均距离最小的两个簇。当使用ward连接时，被合并的是使得SSE增量最小的两个簇。

在本节，我们将重点关注基于全连接方法的凝聚层次聚类，其迭代过程可总结如下：

1）计算得到所有样本间的距离矩阵。

2）将每个数据点看作是一个单独的簇。

3）基于最不相似（距离最远）样本的距离，合并两个最接近的簇。

4）更新相似矩阵（样本间距离矩阵）。

5）重复步骤2到4，直到所有样本都合并到一个簇为止。

现在我们来讨论一下如何计算距离矩阵（步骤1）。在此之前，需要先随机生成一些样本数据用于计算。其中行代表不同的样本（ID值从0到

4），列代表样本的不同特征（X，Y，Z）。

```
>>> import pandas as pd
>>> import numpy as np
>>> np.random.seed(123)
>>> variables = ['X', 'Y', 'Z']
>>> labels = ['ID_0','ID_1','ID_2','ID_3','ID_4']
>>> X = np.random.random_sample([5,3])*10
>>> df = pd.DataFrame(X, columns=variables, index=labels)
>>> df
```

执行上述代码后，可以得到如下距离矩阵：

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

11.2.1　基于距离矩阵进行层次聚类

我们使用SciPy中spatial.distanct子模块下的pdist函数来计算距离矩阵，此矩阵作为层次聚类算法的输入：

```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...                 pdist(df, metric='euclidean')),
...                 columns=labels, index=labels)
>>> row_dist
```

在上述代码中，我们基于样本的特征X、Y和Z，使用欧几里得距离计算了样本间的两两距离。通过将pdist函数的返回值输入到squareform函数中，我们得到了一个记录成对样本间距离的对称矩阵：

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

下面我们使用SciPy中cluster.hierarchy子模块下的linkage函数，此函数以全连接作为距离判定标准，它能够返回一个所谓的关联矩阵（linkage matrix）。

不过在调用linkage函数之前，我们先仔细研究下此函数相关的文档信

息：

```
>>> from scipy.cluster.hierarchy import linkage
>>> help(linkage)
[…]
Parameters:
  y : ndarray
    A condensed or redundant distance matrix. A condensed
    distance matrix is a flat array containing the upper
    triangular of the distance matrix. This is the form
    that pdist returns. Alternatively, a collection of m
    observation vectors in n dimensions may be passed as
    an m by n array.

  method : str, optional
    The linkage algorithm to use. See the Linkage Methods
    section below for full descriptions.

  metric : str, optional
    The distance metric to use. See the distance.pdist
    function for a list of valid distance metrics.

 Returns:
  Z : ndarray
    The hierarchical clustering encoded as a linkage matrix.
[…]
```

从对函数的描述中可知：我们可以将由pdist函数得到的稠密矩阵（上三角）作为输入项。或者，将初始化的欧几里得距离矩阵（将矩阵参数项的值设定为euclidean）作为linkage的输入。不过，我们不应使用前面提及的用squareform函数得到的距离矩阵，因为这会生成与预期不同的距离值。综合来讲，这里可能出现三种不同的情况：

·错误的方法：在本方法中，我们使用了通过squareform函数得到的距离矩阵，代码如下：

```
>>> from scipy.cluster.hierarchy import linkage
>>> row_clusters = linkage(row_dist,
...                        method='complete',
...                        metric='euclidean')
```

·正确的方法：在本方法中，我们使用了稠密距离矩阵，代码如下：

```
>>> row_clusters = linkage(pdist(df, metric='euclidean'),
...                        method='complete')
```

·正确的方法：在本方法中，我们以矩阵格式的示例数据作为输入，代码如下：

```
>>> row_clusters = linkage(df.values,
...                        method='complete',
...                        metric='euclidean')
```

为了更进一步分析聚类结果，我们通过下面的方式将数据转换为pandas的DataFrame格式（最好在IPython Notebook中使用）：

```
>>> pd.DataFrame(row_clusters,
...      columns=['row label 1',
...               'row label 2',
...               'distance',
...               'no. of items in clust.'],
...      index=['cluster %d' %(i+1) for i in
...             range(row_clusters.shape[0])])
```

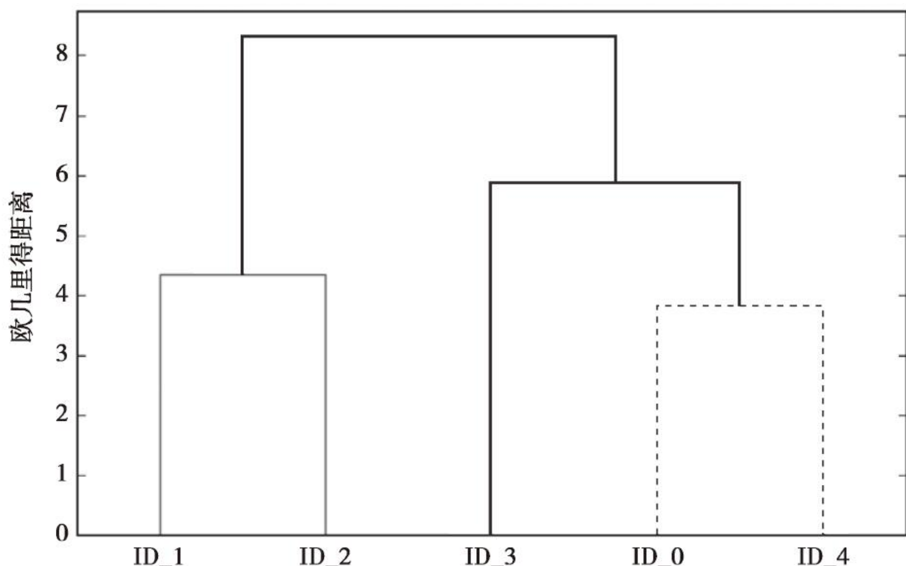
如下表所示，关联矩阵包含多行，每行代表一次簇的合并。矩阵的第一列和第二列分别表示每个簇中最不相似（距离最远）的样本，第三列为这些样本间的距离，最后一列为每个簇中样本的数量。

	row label 1	row label 2	distance	no. of items in clust.
cluster 1	0	4	3.835396	2
cluster 2	1	2	4.347073	2
cluster 3	3	5	5.899885	3
cluster 4	6	7	8.316594	5

现在我们已经完成了关联矩阵的计算，下面采用树状图的形式对聚类结果进行可视化展示：

```
>>> from scipy.cluster.hierarchy import dendrogram
# make dendrogram black (part 1/2)
# from scipy.cluster.hierarchy import set_link_color_palette
# set_link_color_palette(['black'])
>>> row_dendr = dendrogram(row_clusters,
...                        labels=labels,
...                        # make dendrogram black (part 2/2)
...                        # color_threshold=np.inf
...                        )
>>> plt.tight_layout()
>>> plt.ylabel('Euclidean distance')
>>> plt.show()
```

如果是通过执行上述代码生成的图像，或者阅读本书的电子版本，会发现树状图的分支使用了不同的颜色。着色方案来自matplotlib的一个色彩列表，它基于树状图中的距离阈值循环生成不同颜色。例如，通过删除上述代码中的相关注释符，就可使用黑色来绘制此树状图。



此树状图描述了采用凝聚层次聚类合并生成不同簇的过程。例如，从图中可见，首先是ID_0和ID_4合并，接下来是ID_1和ID_2合并，也就是基于欧几里德距离矩阵，选择最不相似的样本进行合并。

11.2.2　树状图与热度图的关联

在实际应用中，层次聚类的树状图通常与热度图（heat map）结合使用，这样我们可以使用不同的颜色来代表样本矩阵中的独立值。在本节中，我们讨论如何将树状图附加到热度图上，并同时显示在一行上。

不过，将树状图与热度图结合还是需要一点小技巧的，我们将逐步介绍这一步骤：

1）创建一个figure对象，并通过add_axes属性来设定x轴位置、y轴位置，以及树状图的宽度和高度。此外，我们沿逆时针方向将树状图旋转90度，代码如下：

```
>>> fig = plt.figure(figsize=(8,8))
>>> axd = fig.add_axes([0.09,0.1,0.2,0.6])
>>> row_dendr = dendrogram(row_clusters, orientation='right')
```

2）接下来，我们根据树状图对象中的簇类标重排初始化数据框（DataFrame）对象中的数据，它本质上是一个Python字典，可通过leaves键访问得到，代码如下：

```
>>> df_rowclust = df.ix[row_dendr['leaves'][::-1]]
```

3）基于重排后的数据框（DataFrame）数据，在树状图的右侧绘制热度图：

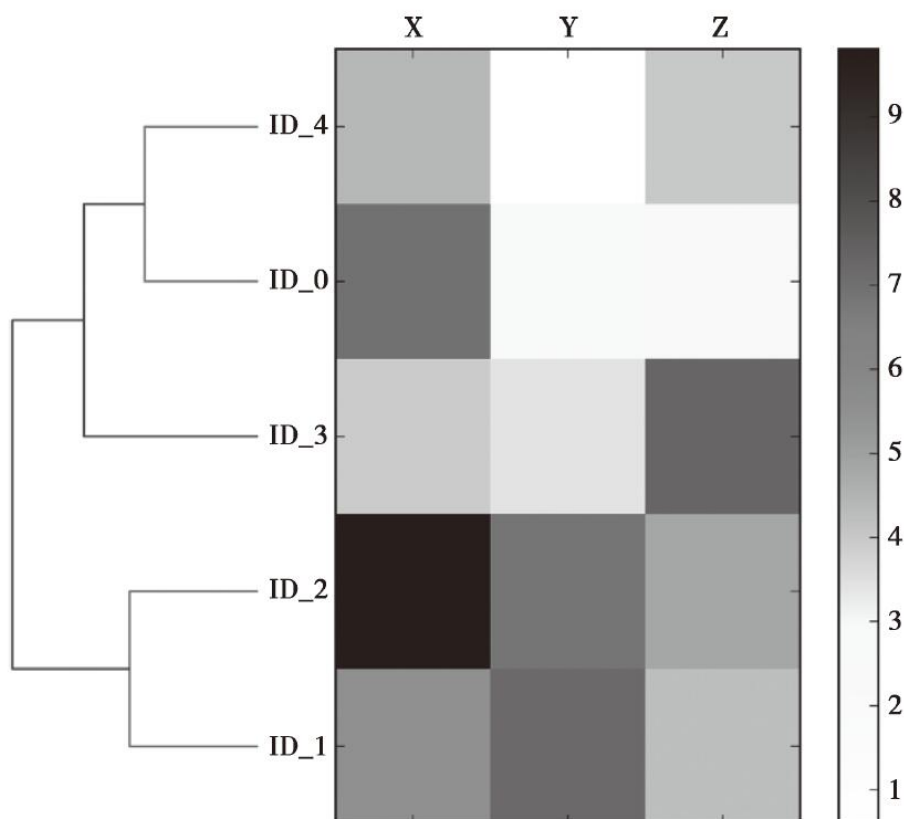
```
>>> axm = fig.add_axes([0.23,0.1,0.6,0.6])
>>> cax = axm.matshow(df_rowclust,
...                   interpolation='nearest', cmap='hot_r')
```

4）最后，为了美化效果，我们删除了坐标轴标记，并将坐标轴的刻度隐藏。此外，我们还加入了色条，并分别在x和y轴上显示特征名和样本ID名称。代码如下：

```
>>> axd.set_xticks([])
>>> axd.set_yticks([])
>>> for i in axd.spines.values():

...     i.set_visible(False)
>>> fig.colorbar(cax)
>>> axm.set_xticklabels([''] + list(df_rowclust.columns))
>>> axm.set_yticklabels([''] + list(df_rowclust.index))
>>> plt.show()
```

通过上述步骤，便可看到热度图和树状图并列显示的图像：



如上图所示，热度图中的行反映了树状图中样本聚类的情况。除了简单的树状图外，热度图中用颜色代表的各样本及其特征为我们提供了关于数据集的一个良好的概括。

11.2.3　通过scikit-learn进行凝聚聚类

在本节中，我们将使用scikit-learn进行基于凝聚的层次聚类。不过，scikit-learn中已经实现了一个AgglomerativeClustering类，它允许我们选择待返回簇的数量。当我们想要对层次聚类树进行剪枝时，这个功能是非常有用的。将参数n_cluster的值设定为2，我们可以采用与前面相同的完全连接方法，基于欧几里得距离矩阵，将样本划分为两个簇：

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> ac = AgglomerativeClustering(n_clusters=2,
...                              affinity='euclidean',
...                              linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print('Cluster labels: %s' % labels)
Cluster labels: [0 1 1 0 0]
```

通过对簇类标的预测结果进行分析，我们可以看出：第一、第四、第五个（ID_0，ID_3和ID_4）样本被划分至第一个簇（0），样本ID_1和ID_2被划分至第二个簇（1），这与此前通过树状图得到的结果一致。

11.3　使用DBSCAN划分高密度区域

尽管无法在本章中介绍大量的不同的聚类算法，但我们至少可以再介绍另一种聚类方法：（包含噪声情况下）基于密度空间的聚类算法（Density-based Spatial Clustering of Applications with Noise，DBSCAN）。在DBSACN中，密度被定义为指定半径ε范围内样本点的数量。

在DBSCAN中，基于以下标准，每个样本（点）都被赋予一个特殊的标签：

·如果在一个点周边的指定半径内，其他样本点的数量不小于指定数量（MinPts），则此样本点称为核心点（core point）。

·在指定半径ε内，如果一个点的邻居点少于MinPts个，但是却包含一个核心点，则此点称为边界点（border point）。

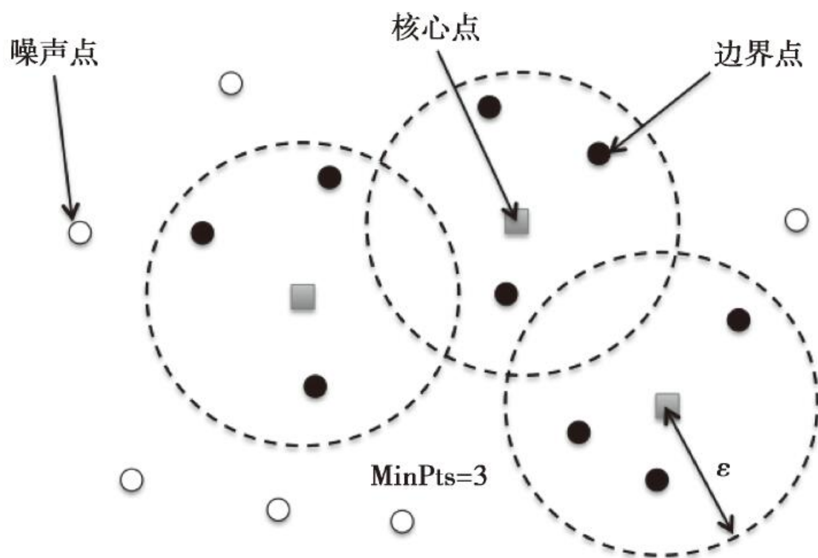
·除了核心点和边界点外的其他样本点称为噪声点（noise point）。

完成对核心点、边界点和噪声点的标记后，DBSCAN算法可总结为两个简单的步骤：

1）基于每个核心点或者一组相连的核心点（如果核心点的距离很近，则将其看作是相连的）形成一个单独的簇。

2）将每个边界点划分到其对应核心点所在的簇中。

在实现具体算法之前，为了让读者对DBSCAN有个更好的直观认识，我们通过下图来了解一下核心点、边界点和噪声点：



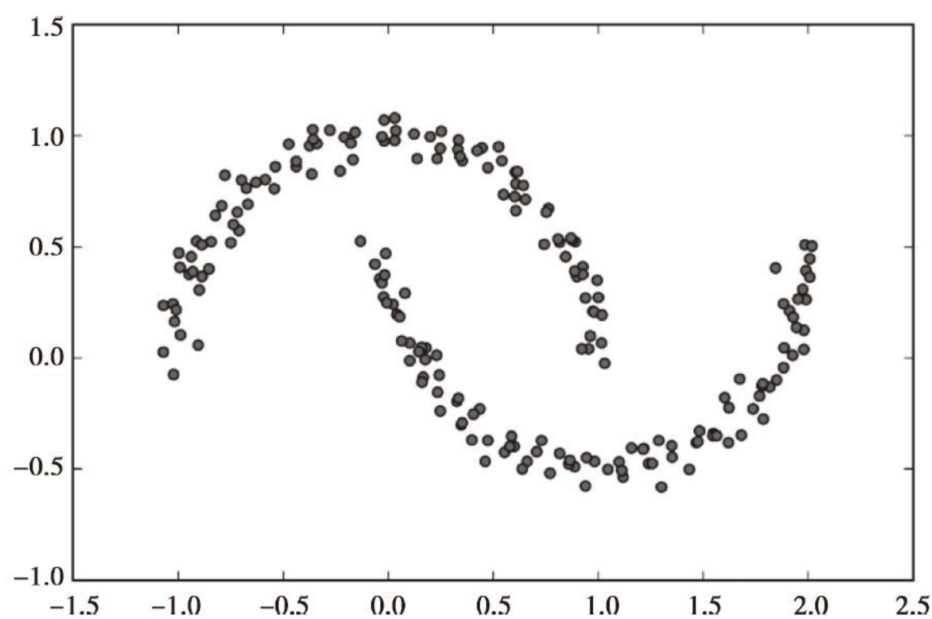
与k-means算法不同，DBSCAN的簇空间不一定是球状的，这也是此算法的优势之一。此外，不同于k-means和层次聚类，由于DBSCAN可以识别并移除噪声点，因此它不一定会将所有的样本点都划分到某一簇中。

为了给出一个更能说明问题的例子，我们创建一个半月形结构的数据集，以对k-means聚类、层次聚类和DBSCAN聚类进行比较：

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=200,
...                   noise=0.05,
...                   random_state=0)
>>> plt.scatter(X[:,0], X[:,1])
>>> plt.show()
```

由结果图像可见，数据集明显地被划分为两个半月形的分组，每个分

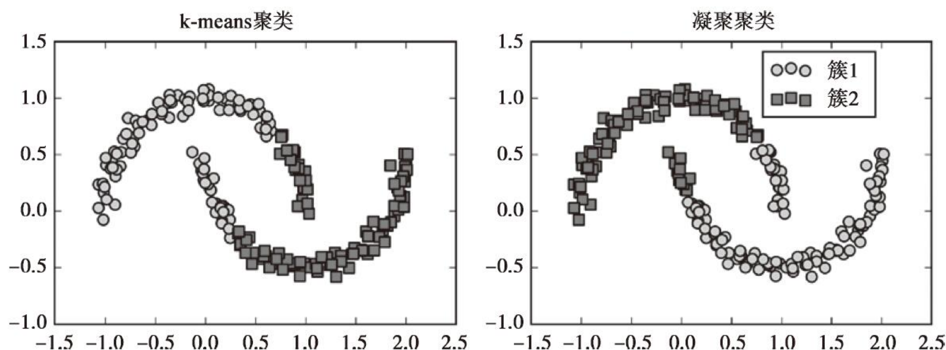
组包含100个样本点：



我们首先使用前面讨论过的k-means算法和基于全连接的层次聚类算法，看它们是否能够成功识别出半月形的簇。代码如下：

```
>>> f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8,3))
>>> km = KMeans(n_clusters=2,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> ax1.scatter(X[y_km==0,0],
...             X[y_km==0,1],
...             c='lightblue',
...             marker='o',
...             s=40,
...             label='cluster 1')
>>> ax1.scatter(X[y_km==1,0],
...             X[y_km==1,1],
...             c='red',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> ax1.set_title('K-means clustering')
>>> ac = AgglomerativeClustering(n_clusters=2,
...                              affinity='euclidean',
...                              linkage='complete')
>>> y_ac = ac.fit_predict(X)
>>> ax2.scatter(X[y_ac==0,0],
...             X[y_ac==0,1],
...             c='lightblue',
...             marker='o',
...             s=40,
...             label='cluster 1')
>>> ax2.scatter(X[y_ac==1,0],
...             X[y_ac==1,1],
...             c='red',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> ax2.set_title('Agglomerative clustering')
>>> plt.legend()
>>> plt.show()
```

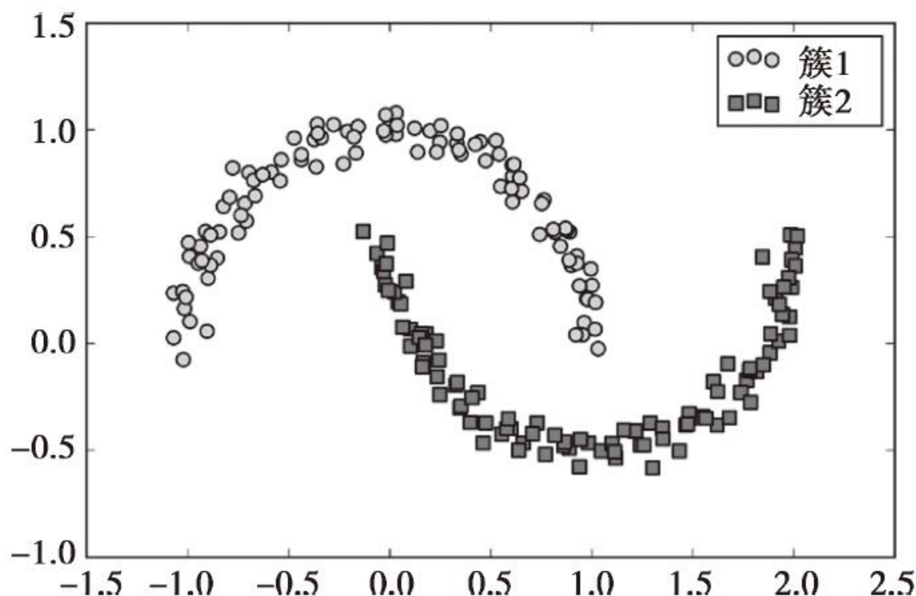
从聚类结果图像可见，k-means算法无法将两个簇分开，而这种复杂形状的数据对层次聚类算法来说也是一种挑战：



最后，我们试一下DBSCAN算法在此数据集上的效果，看其是否能使用基于密度的方法发现两个半月形的簇：

```
>>> from sklearn.cluster import DBSCAN
>>> db = DBSCAN(eps=0.2,
...             min_samples=5,
...             metric='euclidean')
>>> y_db = db.fit_predict(X)
>>> plt.scatter(X[y_db==0,0],
...             X[y_db==0,1],
...             c='lightblue',
...             marker='o',
...             s=40,
...             label='cluster 1')
>>> plt.scatter(X[y_db==1,0],
...             X[y_db==1,1],
...             c='red',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> plt.legend()
>>> plt.show()
```

DBSCAN算法可以成功地对半月形数据进行划分，这也是DBSCAN的优势之一（可对任意形状的数据进行聚类）。



同时，我们也应注意到DBSCAN算法的一些缺点。对于一个给定样本数量的训练数据集，随着数据集中特征数量的增加，维度灾难（curse of dimensionality）的负面影响会随之递增。在使用欧几里得距离度量时，此问题尤为突出。不过，并不是只有DBSCAN算法面临维度灾难问题，使用欧几里得距离度量的其他聚类算法，如k-means和层次聚类算法也都面临此问题。此外，为了能够生成更优的聚类结果，需要对DBSCAN中的两个超参（MinPts和ε）进行优化。如果数据集中的密度差异相对较大，则找到合适的MintPts及的组合较为困难。

到目前为止，我们介绍了三种最基本的聚类方法：k-means基于原型的聚类、凝聚层次聚类、使用DBSCAN基于密度的聚类。在此，有必要提一下另一种更为先进的聚类方法：图聚类。而图聚类系列中最为突出的方法应该是谱聚类算法。虽然实现谱聚类有多种不同的方法，但它们的

共同之处在于：均使用基于相似矩阵的特征向量来获得簇间的关系。由于谱聚类超出了本书的讲解范围，有兴趣的读者可以通过以下链接了解详细内容：http://arxiv.org/pdf/0711.0189v1.pdf（U.Von Luxburg.A Tutorial on Spectral Clustering.Statistics and computing,17（4）:395-416,2007）。

请注意，在实际应用中，对于给定的数据集，往往不太确定选择哪种算法是最为适宜的，特别是面对难以或者无法进行可视化处理的高维数据集。此外，需要特别强调的是，一个好的聚类并不仅仅依赖于算法及其超参的调整。相反，选择合适的距离度量标准和专业领域知识在实验设定时的应用可能会更有帮助。

本章小结

本章中，读者学习了三种不同的聚类算法，它们可以帮助我们发现隐藏在数据背后的结构或者信息。本章开始就介绍了基于原型的k-means算法，此算法可以基于指定数量的簇中心，将样本划分为球形的簇。由于聚类是一种无监督方法，我们无法奢求根据真实的类标来衡量模型的性能。因此，我们使用数据内在有用的性能指标（如肘方法，或者轮廓分析）来尝试对聚类的质量进行量化评定。

接下来，我们学习了另外一种聚类方法：凝聚层次聚类。层次聚类不需要事先指定簇的数量，而且聚类的结果可以通过树状图进行可视化展示，这有助于分析和解释聚类结果。本章介绍的最后一个算法是DBSCAB，此算法基于样本的密度对其进行分组，并且它可以处理异常值以及识别非球型簇。

在学习完无监督学习知识后，是时候介绍一些监督学习领域最令人兴奋的机器学习算法了：多层人工神经网络。随着近年来的复兴，神经网络又一次成为机器学习研究领域的热门话题。借助于新近推出的深度学习算法，神经网络被视为适用于图像分类和语音识别等多种复杂任务的最先进方法。在第12章，我们将从零开始构建多层神经网络。在第13章，我们将介绍一种功能强大的第三方库，它能够帮助我们高效地训练复杂的神经网络。

第12章　使用人工神经网络识别图像

众所周知，深度学习正逐渐获得越来越多的关注，并且毫无疑问成为机器学习领域最热门的话题。深度学习可以被看作是一组算法的集合，这些算法能够高效地进行多层人工神经网络训练。在本章，读者将学习人工神经网络的基本概念，并且接触到新近基于Python开发的深度学习库，从而更进一步去探索机器学习研究领域中这一最为有趣的内容。

本章将涵盖如下主题：

·从概念层次理解多层神经网络

·训练用于图像分类的神经网络

·实现强大的反向传播算法

·调试已实现的神经网络

12.1 使用人工神经网络对复杂函数建模

我们在第2章中从人工神经元入手，开始了机器学习算法的探索。对于本章中将要讨论的多层人工神经网络来说，人工神经元是其构建的基石。人工神经网络的基本概念是建立在对人脑如何应对复杂问题的假想和模型构建上的。人工神经网络在近几年得到了普及，对它的研究最早可追溯到20世纪40年代，Warren McCulloch和Walter Pitt第一个给出了关于神经元如何工作的描述。Rosenblatt在20世纪50年代第一个实现了基于麦卡洛克-皮特（McCulloch-Pitt）神经元模型的感知器算法，在之后的几十年中，由于没有好的训练多层神经网络的算法，许多机器学习的研究人员和从业者逐渐对它失去了兴趣。直到1986年，D.E.Rumelhart、G.E.Hinton和R.J.Williams经过潜心研究，提出并推广了反向传播算法之后，才重新引起了人们对神经网络的重视。关于此算法的详细内容我们将在本章后续做进一步讨论[1]。

在过去的十年中，神经网络研究领域的许多重大突破成就了当前的深度学习算法，此算法可以通过无类标数据训练的深度神经网络（多层神经网络）进行特征检测。神经网络不仅仅是学术领域的一个热门话题，连Facebook、微软及谷歌等大型科技公司都在人工神经网络和深度学习研究领域进行了大量的投入。时至今日，由于能够解决图像和语音识别等复杂问题，由深度学习算法所驱动的复杂神经网络被认为是最前沿的研究成果。我们日常生活中深度学习的常见例子有谷歌图片搜索和谷歌翻译，谷歌翻译作为一款智能手机应用，能够自动识别图片中的文字，并将其实时翻译为20多种语言[2]。

当前一些主要的科技公司正在积极开发更多有趣的深度神经网络应用，如Facebook公司开发的给图片打标签的DeepFace[3]，百度公司的DeepSpeech，它可以处理普通话语音查询[4]。此外，制药行业近期也开始将深度学习技术用于新药研发和毒性预测，研究表明，这些新技术的性能远超过传统的虚拟筛选方法[5]。

[1] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986). Learning Representations by Backpropagating Errors. Nature 323 (6088): 533-536.

[2] http://googleresearch.blogspot.com/2015/07/how-google-translate-squeezes-deep.html.

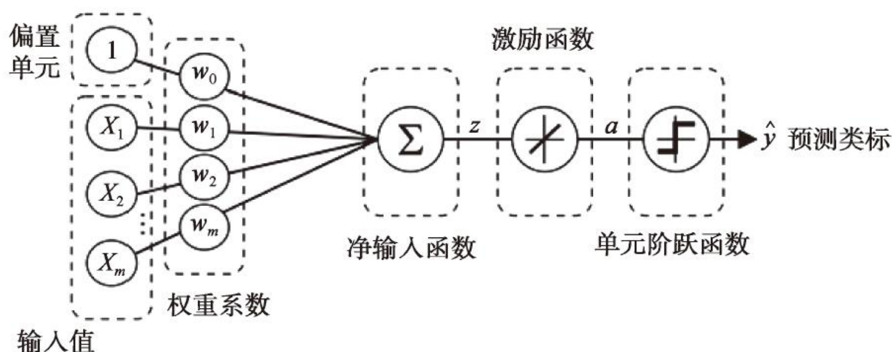
[3] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. DeepFace: Closing the gap to human-level performance in face verification. In Computer Vision and Pattern Recognition CVPR, 2014 IEEE Conference, pages 1701-1708.

[4] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A.Coates, et al. DeepSpeech: Scaling up end-to-end speech recognition. arXiv preprint arXiv:1412.5567, 2014.

[5] T. Unterthiner, A. Mayr, G. Klambauer, and S. Hochreiter. Toxicity prediction using deep learning. arXiv preprint arXiv:1503.01445, 2015.

12.1.1　单层神经网络回顾

本章涉及的内容主要是多层神经网络，包括其工作原理，以及如何对其进行训练以解决复杂问题等。不过在深入讨论多层神经网络结构之前，我们先简要回顾下第2章中介绍的单层神经网络的相关概念，即如下图所示的自适应线性神经元（Adaline）算法：



在第2章中，我们实现了用于二类别分类的Adaline算法，并通过梯度下降优化算法来学习模型的权重系数。训练集上的每一次迭代，我们使用如下更新规则来更新权重向量w：

$$\boldsymbol{w}:=\boldsymbol{w}+\Delta\boldsymbol{w}，\text{其中}\ \Delta\boldsymbol{w}=-\eta\nabla J(\boldsymbol{w})$$

换句话说，我们基于整个训练数据集来计算梯度，并沿着与梯度▿J（w）相反的方向前进以更新模型的权重。为了找到模型的最优权重，我们将待优化的目标函数定义为误差平方和（SSE）代价函数J（w）。此外，我们还为梯度增加了一个经过精心挑选的因子：学习速率η，在学习过程中用于权衡学习速度和代价函数全局最优点之间的关系。

在梯度下降优化过程中，我们在每次迭代后同时更新所有权重，并将权重向量w中各权值w_j的偏导定义为：

$$\frac{\partial}{\partial w_j}J(\boldsymbol{w})=\sum_i(y^{(i)}-a^{(i)})x_j^{(i)}$$

其中，$y^{(i)}$为特定样本$x^{(i)}$的真实类标，$a^{(i)}$为神经元的激励，它是Adaline在特定情形下的线性函数。此外，我们将激励函数φ（·）定义为：

$$\phi(z)=z=a$$

其中，净输入z是输入与权重的线性组合，用于连接输入和输出层：

$$z=\sum_j w_j x_j=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$$

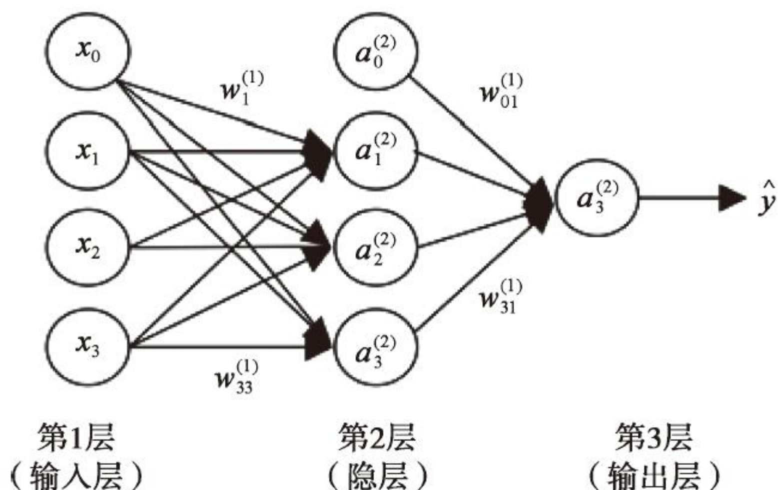
使用激励φ（z）来计算梯度更新时，我们定义了一个阈值函数（单位阶跃函数）将连续的输出值转换为二类别分类的预测类标：

$$\hat{y}=\begin{cases}1 & 若\ g(z)\geqslant 0\\ -1 & 其他\end{cases}$$

请注意：尽管Adaline包含一个输入层和一个输出层，但由于两层之间只有单一的网络连接，因此仍旧称为单层神经网络。

12.1.2　多层神经网络架构简介

在本节，我们将看到如何将多个单独的神经元连接为一个多层前馈神经网络（multi-layer feedforward neural network）。这种特殊类型的网络也称作多层感知器（multi-layer perceptron，MLP）。下图解释了三层MLP的概念：一个输入层、一个隐层，以及一个输出层。隐层的所有单元完全连接到输入层上，同时输出层的单元也完全连接到了隐层中。如果网络中包含不止一个隐层，我们则称其为深度人工神经网络。



我们可以向MLP中加入任意数量的隐层来创建更深层的网络架构。实际上，可以将神经网络中的隐层数量及各单元看作是额外的超参，可以使用第6章中介绍过的交叉验证针对特定问题对它们进行优化。

不过，随着增加到网络中的隐层越来越多，通过反向传播算法计算得到的梯度误差也将变得越来越小。这个接近于不存在的梯度问题使得模型

的学习非常具有挑战性。因此，特别发展出了针对此类深层神经网络架构的预处理算法，亦即所谓的深度学习。

如前面的图所示，我们将第l层中第i个激励单元记为$a_i^{(l)}$，同时激励单元$a_0^{(1)}$和$a_0^{(2)}$为偏置单元（bias unit），我们均设定为1。输入层各单元的激励为输入加上偏置单元：

$$a^{(1)}=\begin{bmatrix} a_0^{(1)} \\ a_1^{(2)} \\ \vdots \\ a_m^{(1)} \end{bmatrix}=\begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}$$

对于l层中的各单元，均通过一个权重系数连接到l+1层中的所有单元上。例如，连接第l层中第k个单元与第l+1层中第j个单元的连接可记作$w_{j,k}^{(l)}$。请注意，$x_m^{(i)}$中的上标i代表是第i个样本，而不是第i层。为了简洁起见，后续段落中我们将忽略上标i。

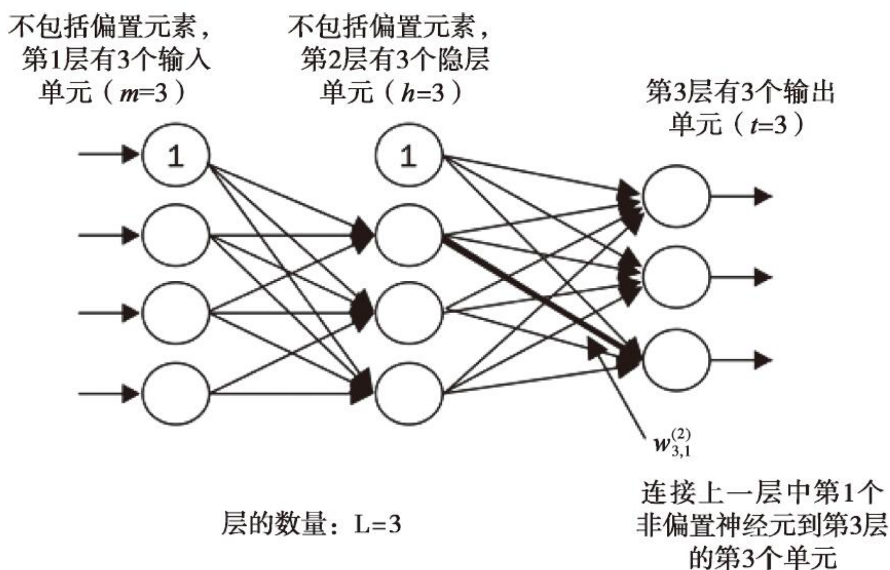
虽然对于二类别分类来说，输出层包含一个激励单元就已足够，但在前面的图中我们可以发现一个更加通用的神经网络形式，通过一对多技术（One-vs-ALL，OvA），可以将它推广到多类别分类上。为了更好理解它是如何工作的，请回忆一下我们在第4章中介绍的使用独热（one-hot）法来表示分类变量。例如，我们可以将鸢尾花数据集中的三个类标（0=Setosa,1=Versicolor,2=Virginica）表示如下：

$$0=\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},\ 1=\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},\ 2=\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

通过独热向量表示方法，我们可以处理数据集中包含任意多个类别的

分类任务。

如果读者是第一次接触到神经网络表示方法，起初可能会对术语符号的标记（上标和下标）感到困惑。也许会疑惑为什么我们不使用$w_{i,k}^{(l)}$，而是使用$w_{j,k}^{(l)}$来表示连接第l层中第k个单元和第l＋1层中第j个单元的权重系数。这些乍看好像很古怪的内容在后续小节中，当使用向量来表示神经网络时将会非常有用。例如，我们将使用矩阵$\boldsymbol{W}^{(i)} \in \mathbb{R}^{h \times [m+1]}$来表示连接输入层和隐层之间的权重，其中h为隐层的数量，而m＋1为隐层中节点及偏置节点数量之和。真正理解并消化这种标识方法对于学习本章后续内容至关重要，我们通过下图来解释一下前面介绍过的3-4-3多层感知器：



12.1.3　通过正向传播构造神经网络

本节中，我们将使用正向传播（forward propagation）来计算多层感知器（MLP）模型的输出。为理解正向传播是如何通过学习来拟合多层感知器模型，我们将多层感知器的学习过程总结为三个简单步骤：

1）从输入层开始，通过网络向前传播（也就是正向传播）训练数据中的模式，以生成输出。

2）基于网络的输出，通过一个代价函数（稍后将有介绍）计算所需最小化的误差。

3）反向传播误差，计算其对于网络中每个权重的导数，并更新模型。

最终，通过对多层感知器模型权重的多次迭代和学习，我们使用正向传播来计算网络的输出，并使用阈值函数获得独热法所表示的预测类标，独热表示法介绍详见上节。

现在，让我们根据正向传播算法逐步从训练数据的模式中生成一个输出。由于隐层每个节点均完全连接到所有输入层节点，我们首先通过以下公式计算$a_1^{(2)}$的激励：

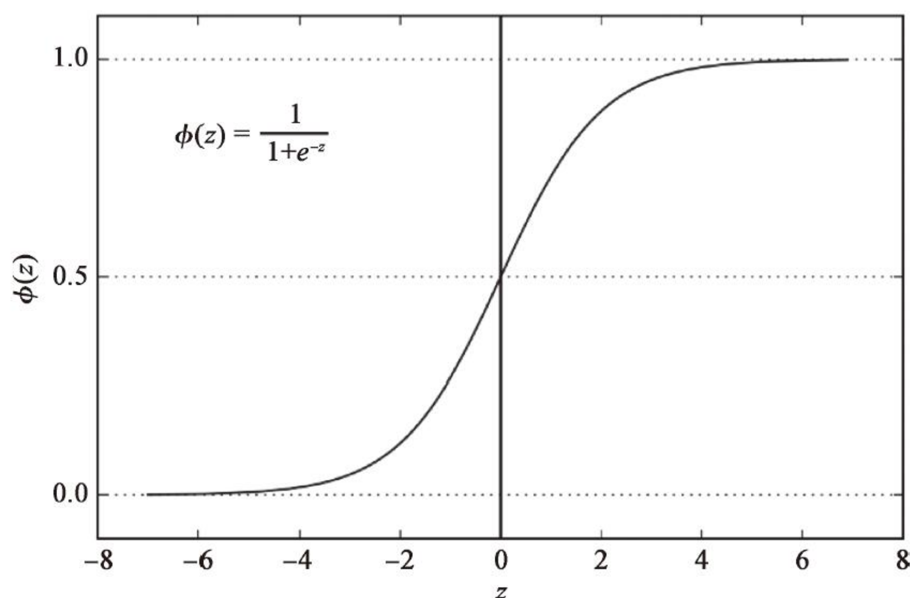
$$z_1^{(2)}=a_0^{(1)}w_{1,0}^{(1)}+a_0^{(1)}w_{1,1}^{(1)}+\cdots+a_m^{(1)}w_{1,m}^{(1)}$$
$$a_1^{(2)}=\phi(z_1^{(2)})$$

其中，$z_1^{(2)}$为净输入，φ（·）为激励函数，此函数必须是可微的，以方

便根据梯度方法学习得到连接神经元的权重。为了解决图像分类等复杂问题，我们需要在多层感知器模型中使用非线性激励函数，例如，在第3章的逻辑斯谛回归中所使用的sigmoid激励函数：

$$\phi(z)=\frac{1}{1+e^{-z}}$$

我们应该还记得，sigmoid函数的图像为S型曲线，它可以将净输入映射到一个介于[0,1]区间的逻辑斯谛分布上去，分布的原点为z = 0.5处，如下图所示：



多层感知器是一个典型的前馈人工神经网络。与本章后续要讨论的递归神经网络不同，此处的前馈是指每一层的输出都直接作为下一层的输入。由于此网络架构中，人工神经元是典型的sigmoid单元，而不是感知器，因此多层感知器这个词听起来有些不够贴切。直观上说，我们可以将

多层感知器中的神经元看作是返回值位于[0,1]连续区间上的逻辑斯谛回归单元。

为了提高代码的执行效率和可读性，我们将使用线性代数中的基本概念，并以紧凑的方式实现一个激励单元：借助NumPy使用向量化的代码，而不是使用效率低下的Python多层for循环嵌套：

$$\mathbf{z}^{(2)} = \boldsymbol{W}^{(1)} \boldsymbol{a}^{(1)}$$

$$\boldsymbol{a}^{(2)} = \phi(\mathbf{z}^{(2)})$$

其中，$\boldsymbol{a}^{(1)}$ 是 $[m+1] \times 1$ 维的样本 $\boldsymbol{x}^{(i)}$ 及其偏置单元。$\boldsymbol{W}^{(1)}$ 为 $h \times [m+1]$ 维的权重矩阵，其中 h 为神经网络中隐层的数量。矩阵与向量相乘之后，我们得到了一个 $h \times 1$ 维的净输入向量 $\boldsymbol{z}^{(2)}$，用于计算 $\boldsymbol{a}^{(2)}$（其中，$\boldsymbol{a}^{(2)} \in \mathbb{R}^{h \times 1}$）。此外，我们还可将此运算推广到全部的 n 个训练样本上：

$$\boldsymbol{Z}^{(2)} = \boldsymbol{W}^{(1)} [\boldsymbol{A}^{(1)}]^{\mathrm{T}}$$

其中，$A^{(1)}$是一个$n \times [m+1]$维的矩阵，而上式中两个矩阵相乘的结果为一个$h \times n$维的净输入矩阵z^2。最后，我们将激励函数φ（·）应用于净输入矩阵中的每个值，便为下一层（本例中为输出层）获得一个$h \times n$维的激励矩阵$A^{(2)}$：

$$\boldsymbol{A}^{(2)} = \phi(\boldsymbol{Z}^{(2)})$$

类似地，我们以向量的形式重写输入层的激励：

$$\boldsymbol{Z}^{(3)} = \boldsymbol{W}^{(2)} \boldsymbol{A}^{(2)}$$

这里，我们将一个$t \times h$维的矩阵$W^{(2)}$（t为输出单元的数量）与$h \times n$维的矩阵$A^{(3)}$相乘，得到了$t \times h$维的矩阵$z^{(3)}$（此矩阵中的列对应每个样本的输出）。

最后，通过sigmoid激励函数，我们可以得到神经网络的连续型输

出：

$$\boldsymbol{A}^{(3)}=\phi(\boldsymbol{Z}^{(3)}),\ \boldsymbol{A}^{(3)}\in\mathbb{R}^{t\times n}$$

12.2　手写数字的识别

在上一节中，我们讨论了大量关于神经网络的理论知识，如果读者初次接触此方面的内容，可能会感到有些困惑。在进一步讨论多层感知器模型的权重学习算法（反向传播算法）之前，先将理论学习暂停片刻，看一下神经网络在实际中的应用。

神经网络理论可以说是相当复杂的，因此作者推荐额外的两篇文献，它们更加详细地讨论了本章介绍过的一些概念：

·T.Hastie,J.Friedman,and R.Tibshirani.The Elements of Statistical Learning,Volume 2.Springer,2009.

·C.M.Bishop et al.Pattern Recognition and Machine Learning,Volume 1.Springer New York,2006.

在本节中，通过在MNIST数据集（Mixed National Institute of Standards and Technology数据集的缩写）上对手写数字的识别，来完成我们第一个多层神经网络的训练。MNIST数据集由Yann LeCun等人创建，是机器学习算法中常用的一个基准数据集[1]。

[1] Y.LeCun,L.Bottou,Y.Bengio,and P.Haffner.Gradient-based Learning Applied to Document Recognition.Proceedings of the IEEE,86(11):2278-2324,November 1998.

12.2.1　获取MNIST数据集

MNIST数据集可通过链接http://yann.lecun.com/exdb/mnist/下载，它包含如下四个部分：

·训练集图像：train-images-idx3-ubyte.gz（9.9 MB，解压后47 MB，包含60000个样本）。

·训练集类标：train-labels-idx1-ubyte.gz（29 KB，解压后60 KB，包含60000个类标）。

·测试集图像：t10k-images-idx3-ubyte.gz（1.6 MB，解压后7.8 MB，包含10000个样本）。

·测试集类标：t10k-labels-idx1-ubyte.gz（5 KB，解压后10 KB，包含10000个类标）。

MNIST数据集基于美国国家标准与技术研究院（National Institute of Standards and Technology，NIST）的两个数据集构建而成。训练集中包含250个人的手写数字，其中50%的是高中生，另外50%来自人口调查局。测试集中的数字也是按照相同比例由高中生和人口调查局所抽选人员手写完成。

下载文件后，建议在UNIX/Linux的命令行终端窗口中，在MNIST文件所在目录，按如下命令使用gzip快速解压所下载文件：

```
gzip *ubyte.gz -d
```

如果读者使用的是Windows操作系统，则可以根据个人喜好选择合适的解压工具。数据集中的图像以字节形式存储，接下来，我们将其读入NumPy数组以训练和测试多层感知器模型：

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte'
                                % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte'
                               % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                    dtype=np.uint8).reshape(len(labels), 784)

    return images, labels
```

load_minist函数返回两个数组，第一个为n×m维的NumPy数组（存储图像），其中n为样本数量，m为特征数量。训练数据集和测试数据集分别包含60000和10000个样本。MNIST数据集中的图像均为28×28个像素，每个像素用灰度强度值表示。在此，我们将28×28像素展开为1维行向量，并用此行向量来表示图像数组（每行或者说每个图像包含784个特征）。load_mnist函数返回的第二个数组（类标）包含对应的目标变量，即手写数字对应的类标（整数0～9）。

乍看起来，我们读取图像的方法好像有些奇怪：

```
magic, n = struct.unpack('>II', lbpath.read(8))
labels = np.fromfile(lbpath, dtype=np.int8)
```

为了解这两行代码是如何工作的，我们看一下MNIST网站上关于此数据集的介绍：

[offset]	*[type]*	*[value]*	*[description]*
0000	*32 bit integer*	*0x00000801(2049)*	*magic number (MSB first)*
0004	*32 bit integer*	*60000*	*number of items*
0008	*unsigned byte*	*??*	*label*
0009	*unsigned byte*	*??*	*label*
……..			
xxxx	*unsigned byte*	*??*	*label*

使用前面两行代码，在使用fromfile方法将后续字节读入NumPy数组之前，我们首先从文件缓冲区读入数据集的幻数，它是对文件协议的描述，同时读入的还有条目的数量。传递给struct.unpack函数中fmt参数的实参值：＞II，此实参值包含两部分内容：

·＞：这是代表大端字节序（定义多字节在计算机中的存储顺序），如果读者不熟悉大端字节序和小端字节序，可以参考维基百科中关于字节序的描述（https://en.wikipedia.org/wiki/Endianness）。

·I：代表这是一个无符号整数。

执行下列代码，我们将从解压后MNIST数据集所在目录mnist下读取60000个训练实例和10000个测试样本：

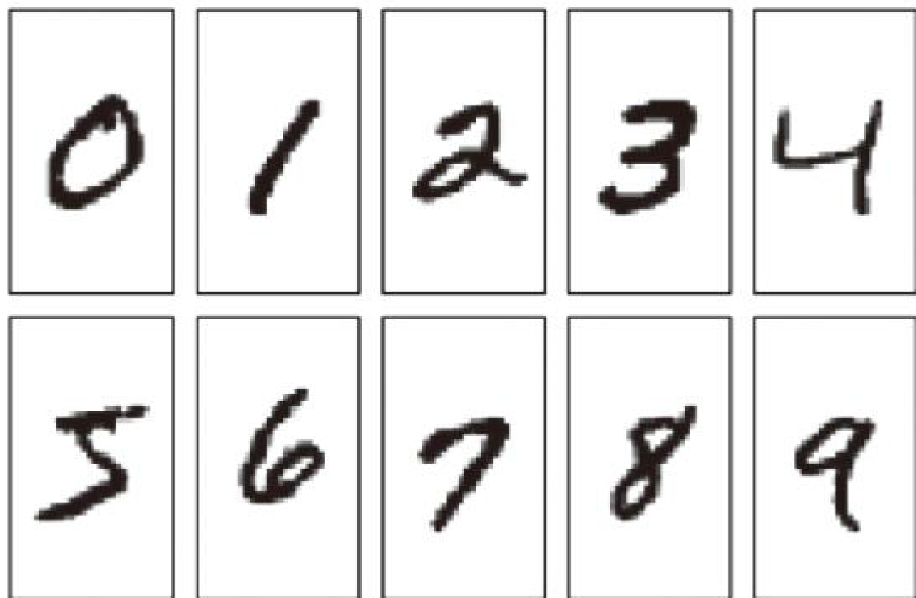
```
>>> X_train, y_train = load_mnist('mnist', kind='train')
>>> print('Rows: %d, columns: %d'
...         % (X_train.shape[0], X_train.shape[1]))
Rows: 60000, columns: 784

>>> X_test, y_test = load_mnist('mnist', kind='t10k')
>>> print('Rows: %d, columns: %d'
...         % (X_test.shape[0], X_test.shape[1]))
Rows: 10000, columns: 784
```

为了解MNIST数据集中图像的样子，我们通过将特征矩阵中的784像素向量还原为28×28图像，并使用matplotlib中的imshow函数将0～9数字的示例进行可视化展示：

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True,
sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X_train[y_train == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys', interpolation='nearest')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

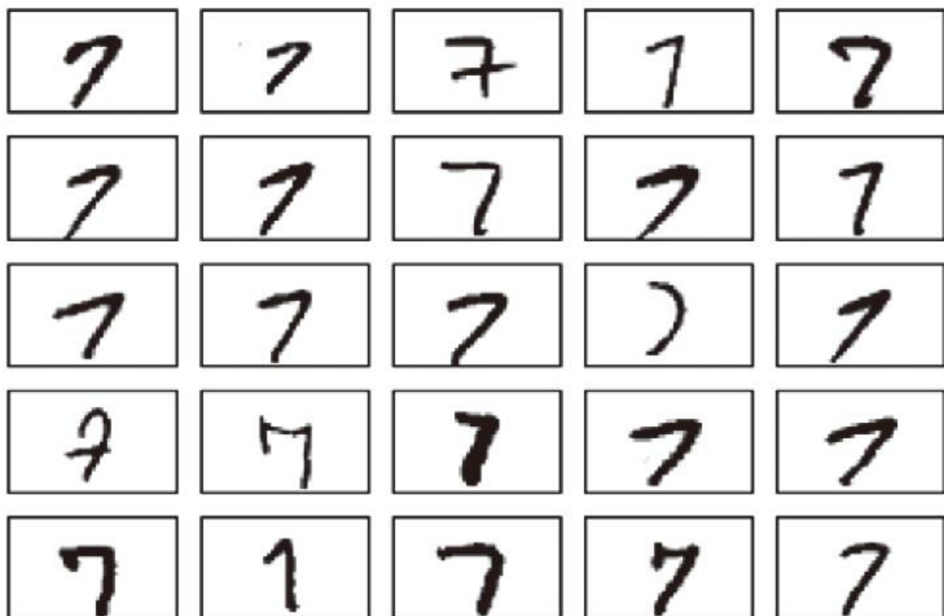
现在我们可以看到，按照2×5方式排列的子图中显示了单个数字的图像：



此外，我们再绘制一下相同数字的多个示例，来看一下这些手写样本之间到底有多大差异：

```
>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = X_train[y_train == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys', interpolation='nearest')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

执行上述代码后，可以看到数字7的前25个不同变体。



我们也可以选择将MNIST图像数据及其对应类标存储为CSV格式的文件，以方便不支持其原始特殊字节格式的程序使用。不过，我们应知道，CSV格式的文件会占用更多的存储空间，具体大小如下：

·train_img.csv:109.5 MB

·train_labels.csv:120 KB

·test_img.csv:18.3 MB

·test_labels:20 KB

在将MNIST数据加载到NumPy数组中后，我们可以在Python中执行如下代码，即可将数据存储为CSV格式文件：

```
>>> np.savetxt('train_img.csv', X_train,
...            fmt='%i', delimiter=',')
>>> np.savetxt('train_labels.csv', y_train,
...            fmt='%i', delimiter=',')
>>> np.savetxt('test_img.csv', X_test,
...            fmt='%i', delimiter=',')
>>> np.savetxt('test_labels.csv', y_test,
...            fmt='%i', delimiter=',')
```

对于已经保存过的CSV格式文件，我们可以使用NumPy的genfromtxt函数对其进行加载：

```
>>> X_train = np.genfromtxt('train_img.csv',
...                         dtype=int, delimiter=',')
>>> y_train = np.genfromtxt('train_labels.csv',
...                         dtype=int, delimiter=',')
>>> X_test = np.genfromtxt('test_img.csv',
...                        dtype=int, delimiter=',')
>>> y_test = np.genfromtxt('test_labels.csv',
...                        dtype=int, delimiter=',')
```

不过，加载CSV格式的MNIST数据需要更长的时间，因此建议读者尽可能使用原始的数据格式。

12.2.2 实现一个多层感知器

本小节中，我们将实现一个包含一个输入层、一个隐层和一个输出层的多层感知器，并用它来识别MNIST数据集中的图像。我们尽可能使代码做到简单易读。不过第一次接触可能会感觉稍许复杂，建议读者从Packt出版社的官网上下载本章示例代码，其中，注释和语法高亮显示使得代码更加易读。如果读者并未在IPython Notebook中运行这些代码，建议将其复制到当前工作目录下的一个Python脚本文件中，如neuralnet.py，进而可以使用如下命令在当前Python工作进程中将其导入：

```
from neuralnet import NeuralNetMLP
```

代码中包含我们尚未讲解的内容，比如反向传播算法，不过其中大部分代码都是基于第2章中自适应线性神经元（Adaline）实现的，因此读起来应该比较熟悉。如果有不理解的代码也不必担心，我们将在本章后续内容中进行讲解。不过，在当前阶段先熟悉代码有利于理解后面理论部分内容。

```
import numpy as np
from scipy.special import expit
import sys

class NeuralNetMLP(object):
    def __init__(self, n_output, n_features, n_hidden=30,
                 l1=0.0, l2=0.0, epochs=500, eta=0.001,
                 alpha=0.0, decrease_const=0.0, shuffle=True,
                 minibatches=1, random_state=None):
        np.random.seed(random_state)
        self.n_output = n_output
        self.n_features = n_features
        self.n_hidden = n_hidden
        self.w1, self.w2 = self._initialize_weights()
        self.l1 = l1
        self.l2 = l2
        self.epochs = epochs
        self.eta = eta
        self.alpha = alpha
        self.decrease_const = decrease_const
        self.shuffle = shuffle
        self.minibatches = minibatches

    def _encode_labels(self, y, k):
        onehot = np.zeros((k, y.shape[0]))
        for idx, val in enumerate(y):
```

```
            onehot[val, idx] = 1.0
        return onehot

    def _initialize_weights(self):
        w1 = np.random.uniform(-1.0, 1.0,
                    size=self.n_hidden*(self.n_features + 1))
        w1 = w1.reshape(self.n_hidden, self.n_features + 1)
        w2 = np.random.uniform(-1.0, 1.0,
                    size=self.n_output*(self.n_hidden + 1))
        w2 = w2.reshape(self.n_output, self.n_hidden + 1)
        return w1, w2

    def _sigmoid(self, z):
        # expit is equivalent to 1.0/(1.0 + np.exp(-z))
        return expit(z)

    def _sigmoid_gradient(self, z):
        sg = self._sigmoid(z)
        return sg * (1 - sg)

    def _add_bias_unit(self, X, how='column'):
        if how == 'column':
            X_new = np.ones((X.shape[0], X.shape[1]+1))
            X_new[:, 1:] = X
        elif how == 'row':
            X_new = np.ones((X.shape[0]+1, X.shape[1]))
            X_new[1:, :] = X
        else:
            raise AttributeError('`how` must be `column` or `row`'
        return X_new

    def _feedforward(self, X, w1, w2):
        a1 = self._add_bias_unit(X, how='column')
        z2 = w1.dot(a1.T)
        a2 = self._sigmoid(z2)
        a2 = self._add_bias_unit(a2, how='row')
        z3 = w2.dot(a2)
        a3 = self._sigmoid(z3)
        return a1, z2, a2, z3, a3

    def _L2_reg(self, lambda_, w1, w2):
        return (lambda_/2.0) * (np.sum(w1[:, 1:] ** 2)\
                + np.sum(w2[:, 1:] ** 2))

    def _L1_reg(self, lambda_, w1, w2):
        return (lambda_/2.0) * (np.abs(w1[:, 1:]).sum()\
                + np.abs(w2[:, 1:]).sum())

    def _get_cost(self, y_enc, output, w1, w2):
        term1 = -y_enc * (np.log(output))
        term2 = (1 - y_enc) * np.log(1 - output)
        cost = np.sum(term1 - term2)
        L1_term = self._L1_reg(self.l1, w1, w2)
        L2_term = self._L2_reg(self.l2, w1, w2)
        cost = cost + L1_term + L2_term
```

```
        return cost

    def _get_gradient(self, a1, a2, a3, z2, y_enc, w1, w2):
        # backpropagation
        sigma3 = a3 - y_enc
        z2 = self._add_bias_unit(z2, how='row')
        sigma2 = w2.T.dot(sigma3) * self._sigmoid_gradient(z2)
        sigma2 = sigma2[1:, :]
        grad1 = sigma2.dot(a1)
        grad2 = sigma3.dot(a2.T)

        # regularize
        grad1[:, 1:] += (w1[:, 1:] * (self.l1 + self.l2))
        grad2[:, 1:] += (w2[:, 1:] * (self.l1 + self.l2))

        return grad1, grad2

    def predict(self, X):
        a1, z2, a2, z3, a3 = self._feedforward(X, self.w1, self.w2)
        y_pred = np.argmax(z3, axis=0)
        return y_pred

    def fit(self, X, y, print_progress=False):
        self.cost_ = []
        X_data, y_data = X.copy(), y.copy()
        y_enc = self._encode_labels(y, self.n_output)

        delta_w1_prev = np.zeros(self.w1.shape)
        delta_w2_prev = np.zeros(self.w2.shape)

        for i in range(self.epochs):

            # adaptive learning rate
            self.eta /= (1 + self.decrease_const*i)

            if print_progress:
                sys.stderr.write(
                        '\rEpoch: %d/%d' % (i+1, self.epochs))
                sys.stderr.flush()

            if self.shuffle:
                idx = np.random.permutation(y_data.shape[0])
                X_data, y_data = X_data[idx], y_data[idx]

            mini = np.array_split(range(
                          y_data.shape[0]), self.minibatches)
            for idx in mini:

                # feedforward
                a1, z2, a2, z3, a3 = self._feedforward(
                                     X[idx], self.w1, self.w2)
                cost = self._get_cost(y_enc=y_enc[:, idx],
                                      output=a3,
                                      w1=self.w1,
                                      w2=self.w2)
                self.cost_.append(cost)
```

```
            # compute gradient via backpropagation
            grad1, grad2 = self._get_gradient(a1=a1, a2=a2,
                                              a3=a3, z2=z2,
                                              y_enc=y_enc[:, idx],
                                              w1=self.w1,
                                              w2=self.w2)

            # update weights
            delta_w1, delta_w2 = self.eta * grad1,\
                                 self.eta * grad2
            self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
            self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
            delta_w1_prev, delta_w2_prev = delta_w1, delta_w2

    return self
```

现在，我们来初始化一个784-50-10的感知器模型，该神经网络包含784个输入单元（n_features），50个隐层单元（n_hidden），以及10个输出单元（n_output）：

```
>>> nn = NeuralNetMLP(n_output=10,
...                   n_features=X_train.shape[1],
...                   n_hidden=50,
...                   l2=0.1,
...                   l1=0.0,
...                   epochs=1000,
...                   eta=0.001,
...                   alpha=0.001,
...                   decrease_const=0.00001,
...                   shuffle=True,
...                   minibatches=50,
...                   random_state=1)
```

读者也许已经注意到，在重复实现多层感知器的同时，我们还额外实现了一些功能，总结如下：

·l2：L2正则化系数λ，用于降低过拟合程度，类似地，l1对应L1正则化参数λ。

·epochs：遍历训练集的次数（迭代次数）。

·eta：学习速率η。

·alpha：动量学习进度的参数，它在上一轮迭代的基础上增加一个因子，用于加快权重更新的学习$\Delta w_t = \eta \nabla J(w_t + \Delta w_{t-1})$（其中，t为当前所在的步骤，也就是当前迭代次数）。

·decrease_const：用于降低自适应学习速率n的常数d，随着迭代次数的增加而随之递减以更好地确保收敛。

·shuffle：在每次迭代前打乱训练集的顺序，以防止算法陷入死循环。

·Minibatches：在每次迭代中，将训练数据划分为k个小的批次，为加速学习的过程，梯度由每个批次分别计算，而不是在整个训练数据集上进行计算。

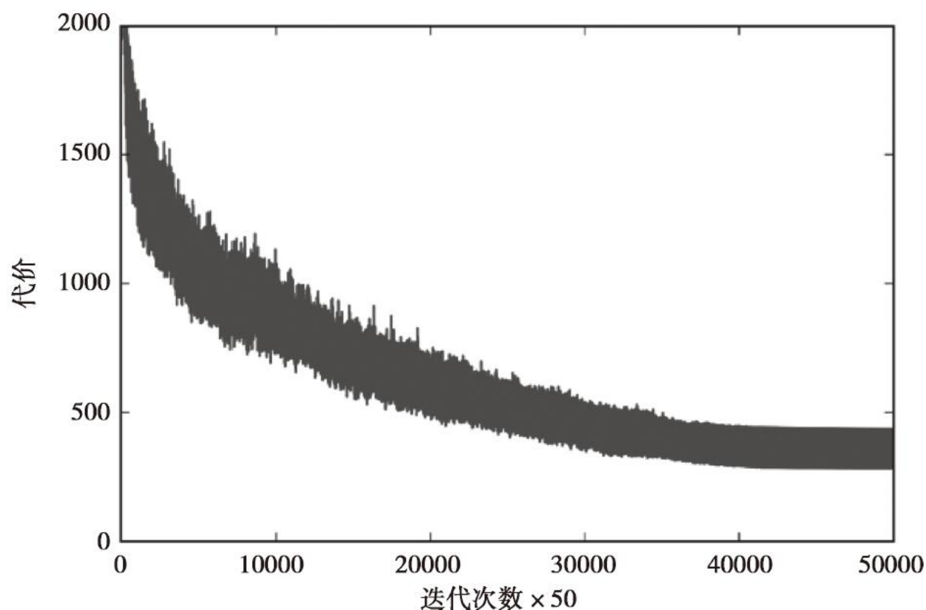
接下来，我们将使用重排后的MNIST训练数据集中的60000个样本来训练多层感知器。执行下列代码前请注意：在当前主流配置的台式计算机上，训练神经网络所需的时间大约为10～30分钟：

```
>>> nn.fit(X_train, y_train, print_progress=True)
Epoch: 1000/1000
```

与前面自适应线性神经元的实现类似，我们使用cost_列表保存了每轮迭代中的代价并且可以进行可视化，以确保优化算法能够收敛。在此，我们仅绘制了50个子批次的每50次迭代的结果（50个子批次×1000次迭代）。代码如下：

```
>>> plt.plot(range(len(nn.cost_)), nn.cost_)
>>> plt.ylim([0, 2000])
>>> plt.ylabel('Cost')
>>> plt.xlabel('Epochs * 50')
>>> plt.tight_layout()
>>> plt.show()
```

通过下图可见，代价函数的图像中有很明显的噪声。这是由于我们使用了随机梯度下降算法的一个变种（子批次学习），来训练神经网络所造成的。

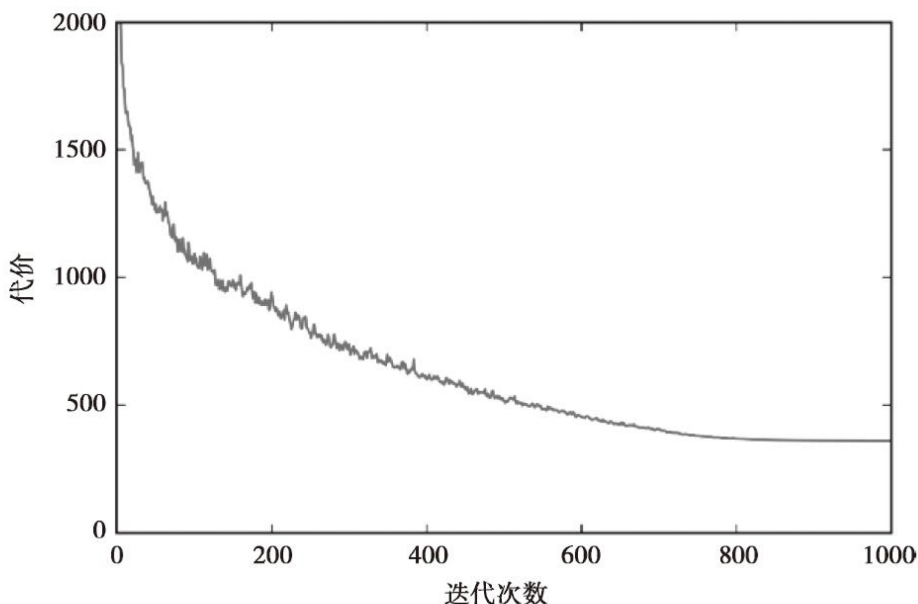


虽然通过上图可以看出，优化算法在经过大约800（40000/50 = 800）轮迭代后收敛，使用所有子批次的平均值，我们绘制出了一个相对平滑的代价函数图像。代码如下：

```
>>> batches = np.array_split(range(len(nn.cost_)), 1000)
>>> cost_ary = np.array(nn.cost_)
>>> cost_avgs = [np.mean(cost_ary[i]) for i in batches]

>>> plt.plot(range(len(cost_avgs)),
...          cost_avgs,
...          color='red')
>>> plt.ylim([0, 2000])
>>> plt.ylabel('Cost')
>>> plt.xlabel('Epochs')
>>> plt.tight_layout()
>>> plt.show()
```

从下图可以清楚地看出，训练算法在经过约800次迭代后随即收敛：



现在，我们通过计算预测精度来评估模型的性能：

```
>>> y_train_pred = nn.predict(X_train)
>>> acc = np.sum(y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Training accuracy: %.2f%%' % (acc * 100))
Training accuracy: 97.74%
```

正如我们所见，模型能够正确识别大部分的训练数字，不过现在还不

知道将其泛化到未知数据上的效果如何？我们来计算一下模型在测试数据集上10000个图像上的准确率：

```
>>> y_test_pred = nn.predict(X_test)
>>> acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
>>> print('Training accuracy: %.2f%%' % (acc * 100))
Test accuracy: 96.18%
```

由于模型在训练集与测试集上的精度仅有微小的差异，我们可以推断，模型对于训练数据仅轻微地过拟合。为了进一步对模型进行调优，我们可以改变隐层单元的数量、正则化参数的值、学习速率、衰减常数的值，或者使用第6章中介绍过的自适应学习等技术（此问题留给读者作为练习作业）。

现在，我们看一下多层感知器难以处理的一些图像：

```
>>> miscl_img = X_test[y_test != y_test_pred][:25]
>>> correct_lab = y_test[y_test != y_test_pred][:25]
>>> miscl_lab= y_test_pred[y_test != y_test_pred][:25]

>>> fig, ax = plt.subplots(nrows=5,
...                         ncols=5,
...                         sharex=True,
...                         sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = miscl_img[i].reshape(28, 28)
...     ax[i].imshow(img,
...                  cmap='Greys',
...                  interpolation='nearest')
...     ax[i].set_title('%d) t: %d p: %d'
...                     % (i+1, correct_lab[i], miscl_lab[i]))
>>> ax[0].set_xticks([])

>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

我们得到一个包含5×5子图矩阵的图像，每个子图标题中的第一个数

字为图像索引，第二个数字为真实的类标（t），第三个数字则是预测的类标（p）。

1) t: 4 p: 0 2) t: 2 p: 4 3) t: 9 p: 3 4) t: 4 p: 6 5) t: 6 p: 0
6) t: 8 p: 4 7) t: 2 p: 0 8) t: 2 p: 7 9) t: 4 p: 9 10) t: 5 p: 3
11) t: 3 p: 7 12) t: 8 p: 2 13) t: 4 p: 6 14) t: 8 p: 7 15) t: 6 p: 0
16) t: 9 p: 8 17) t: 9 p: 3 18) t: 8 p: 2 19) t: 5 p: 3 20) t: 9 p: 4
21) t: 7 p: 3 22) t: 4 p: 9 23) t: 3 p: 7 24) t: 4 p: 6 25) t: 1 p: 8

从上图可以看出，某些图像即便让我们人工去分类也存在一定难度。例如，数字9的下部呈弯钩状，因此被识别成3或者8（参见第3、16和17子图）。

12.3 人工神经网络的训练

我们已经了解了神经网络在的具体应用，并通过学习代码对其如何工作有了基本的认识，现在我们将更深入挖掘一些概念，如用于权值学习的逻辑斯谛代价函数（logistic cost function）和反向传播（backpropagation）算法。

12.3.1　计算逻辑斯谛代价函数

我们在_get_cost方法中实现的逻辑斯谛代价函数其实很简单，它实际上与我们在第3章逻辑斯谛回归那一小节中介绍的代价函数完全一致。

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n} y^{(i)} \log(a^{(i)}) + (1-y^{(i)})\log(1-a^{(i)})$$

其中，a（i）是前向传播过程中，用来计算某层中第i个单元的sigmoid激励函数。

$$a^{(i)} = \phi(z^{(i)})$$

现在，我们加入一个正则化项，它可以帮助我们降低过拟合的程度。回忆一下前面章节中的内容，L2和L1正则化项定义如下（请记住，我们不对偏置单元进行正则化处理）：

$$L2 = \lambda \|\boldsymbol{w}\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2 \quad 且 \quad L1 = \lambda \|\boldsymbol{w}\|_1^1 = \lambda \sum_{j=1}^{m} |w_j|$$

虽然L1和L2正则化均适用于我们的多层感知器模型，为简单起见，我们只关注L2正则化项。不过这些同样的概念也适用于L1正则化项。通过在逻辑斯谛代价函数中加入L2正则化项，我们可得到如下公式：

$$J(\boldsymbol{w}) = \left[\sum_{i=1}^{n} y^{(i)} \log(a^{(i)}) + (1-y^{(i)})\log(1-a^{(i)})\right] + \frac{\lambda}{2} \|\boldsymbol{w}\|_2^2$$

我们已经实现了一个用于多类别分类的多层感知器，它返回一个包含t个元素的输出向量，我们需要将其与使用独热编码表示的t×1维目标向量进行比较。例如，对于一个特定的样本来说，它在第三层的激励和目标类

别（此处为类别2）可能如下所示：

$$a^{(3)}=\begin{bmatrix}0.1\\0.9\\\vdots\\0.3\end{bmatrix},\ y=\begin{bmatrix}0\\1\\\vdots\\0\end{bmatrix}$$

由此，我们需要将逻辑斯谛代价函数运用到网络中的所有激励单元j。因此我们的代价函数（未加入正则化项）变为：

$$J(\boldsymbol{w})=-\sum_{i=1}^{n}\sum_{k=1}^{t}y_j^{(i)}\log(a_j^{(i)})+(1-y_j^{(i)})\log(1-a_j^{(i)})$$

其中，上标i为训练集中特定样本的索引。

下面经泛化的正则化项看起来有点复杂，不过在此我们仅计算l层所有加到第一列权重的总和（不包含偏置项）

$$J(\boldsymbol{w})=-\left[\sum_{i=1}^{n}\sum_{k=1}^{t}\ y_j^{(i)}\log(\phi(z^{(i)})j)+(1-y_j^{(i)})\log(1-\phi(z^{(i)})j)\right]+\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{ul}\sum_{j=1}^{ul+1}(w_{j,t}^{(l)})^2$$

以下公式表示L2罚项：

$$+\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{ul}\sum_{j=1}^{ul+1}(w_{j,t}^{(l)})^2$$

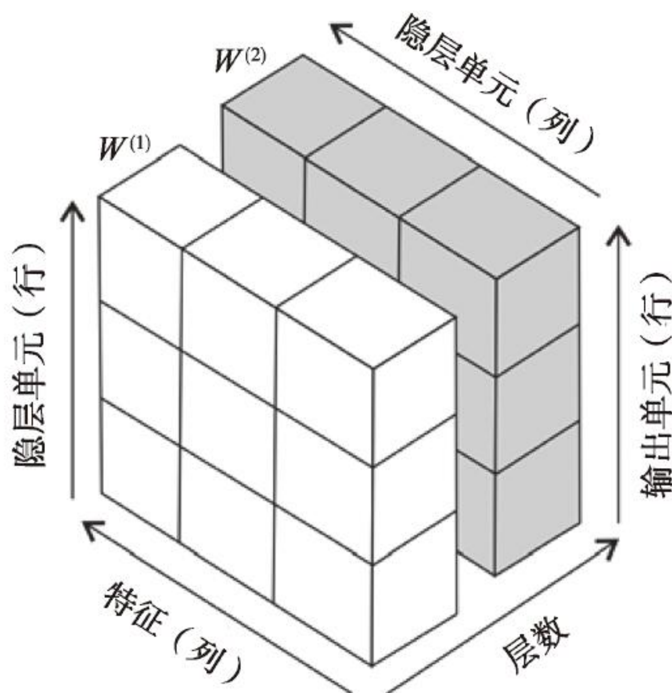
请记住，我们的目标是最小化代价函数J（w）。因此，须计算矩阵W对网络各层中每个权重的偏导：

$$\frac{\partial}{\partial w_{j,i}^{(l)}}J(\boldsymbol{W})$$

在下一节中，我们将讨论反向传播算法，它能够通过计算偏导最小化

代价函数。

请注意，W包含多个矩阵。在仅包含一个隐层单元的多层感知器中，可通过权重矩阵$W^{(1)}$将输入层与隐层相连，而$W^{(2)}$则连接隐层与输出层。下图以一种直观的方式对矩阵W进行可视化展示：



在这个简化图中，矩阵$W^{(1)}$和$W^{(2)}$均具有相同的行数和列数，此情况仅当多层感知器的隐层、输入层、输出层数量相同时才会出现。

如果对此感到困惑，请进入下一节，我们将通过反向传播算法详细讨论$W^{(1)}$和$W^{(2)}$维度方面的内容。

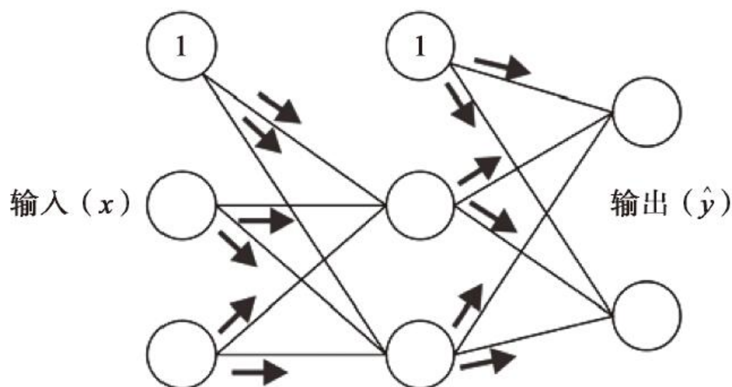
12.3.2　通过反向传播训练神经网络

在本节中，我们将通过讲解反向传播算法中相关数学公式，使读者理解神经网络是如何通过学习高效获得权重的。每个人对数学表示方法的熟练程度不同，因此下面将要介绍的公式可能看起来会有些复杂。许多人喜欢自底向上的方法，通过公式的逐步讲解形成对算法的直观认识。不过，如果你喜欢自顶向下的方法，并且希望在不使用数学符号的情况下了解反向传播算法，建议读者先阅读下一节内容后再来学习本节内容。

在上一节中，我们学习了如何通过最后一层的激励以及目标类标之间的差异来计算代价。现在，我们将了解一下反向传播算法如何更新多层感知器模型的权重，_get_gradient方法已实现了该算法。回忆下本章开始时介绍的内容，我们首先需要通过正向传播来获得输出层的激励，可将其形式化为：

$$\boldsymbol{Z}^{(2)}=\boldsymbol{W}^{(1)}[\boldsymbol{A}^{(1)}]^{\mathrm{T}} \text{（隐层的净输入）}$$
$$\boldsymbol{A}^{(2)}=\phi(\boldsymbol{Z}^{(2)}) \text{（隐层的激励）}$$
$$\boldsymbol{Z}^{(3)}=\boldsymbol{Z}^{(2)}\boldsymbol{A}^{(2)} \text{（输出层的净输出）}$$
$$\boldsymbol{A}^{(3)}=\phi(\boldsymbol{Z}^{(3)}) \text{（输出层的激励）}$$

简单地说，我们只是按照下图所示，通过网络中的连接将输入向前传递：



在反向传播中，错误被从右至左传递。我们首先计算输出层的误差向量：

$$\delta^{(3)}=a^{(3)}-y$$

其中，y为真实类标的向量。

接下来，我们计算隐层的误差项：

$$\boldsymbol{\delta}^{(3)}=(\boldsymbol{W}^{(2)})^{T}\delta^{(2)}\times\frac{\partial\phi(z^{(2)})}{\partial z^{(2)}}$$

其中，$\frac{\partial\phi(z^{(2)})}{\partial z^{(2)}}$为 sigmoid 激励函数的导数，在 _sigmoid_gradient 中已得到实现：

$$\frac{\partial\phi(z)}{\partial z}=(a^{(2)}\times(1-a^{(2)}))$$

请注意：星号（*）在此表示逐元素相乘。

虽然无需太过在意下面的公式，不过读者可能会好奇：激励函数的导数是如何得到的？在此逐步演示求导过程：

$$\phi'(z)=\frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right)=\frac{e^{-z}}{(1+e^{-z})^{2}}=\frac{1+e^{-z}}{(1+e^{-z})^{2}}-\left(\frac{1}{1+e^{-z}}\right)^{2}=\frac{1}{(1+e^{-z})}-\left(\frac{1}{1+e^{-z}}\right)^{2}$$
$$=\phi(z)-(\phi(z))^{2}=\phi(z)(1-\phi(z))=a(1-a)$$

为了更好地理解δ（2）项的计算，我们对此进行更详细的讨论。在前面的公式中，我们将其与t×h维矩阵W（2）的转置（W（2））T相乘，t为输出类标的数量，而h为隐层单元的数量。现在，（W（2））T与t×1维向量δ（2）的乘积为h×t维矩阵。我们将（W（2））Tδ（3）与（a（2）*（1-a（2）））逐项相乘，结果也是一个t×1维的向量。最后，在得到δ后，我们可将代价函数的导数写作：

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(\boldsymbol{W}) = a_j^{(l)} \delta_i^{(l+1)}$$

接下来，要计算偏导，我们需要将l层中的第j个节点的偏导与l＋1层中第i个节点的误差进行累加：

$$\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

请记住，我们需要计算训练集中每个样本的$w_{i,j}^{(l)}$。因此，与前面实现多层感知器的代码类似，使用向量的表达方式会更容易一些：

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} (\boldsymbol{A}^{(l)})^{\mathrm{T}}$$

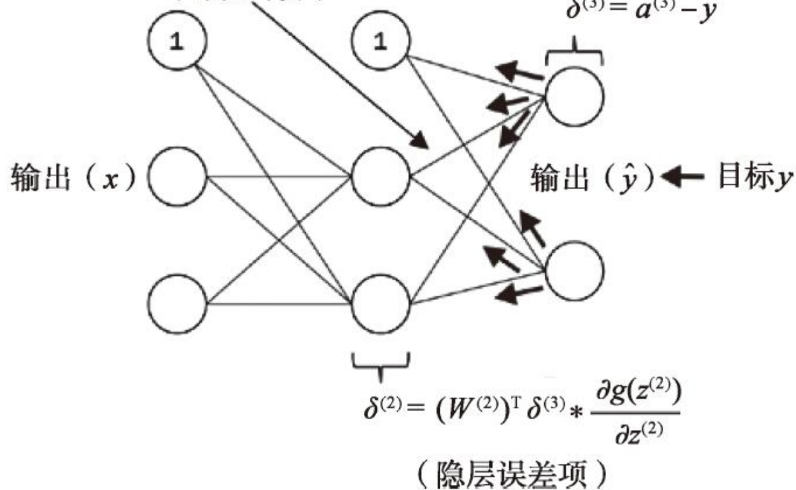
在完成偏导的累加后，我们可以通过下列方式加入正则化项：

$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} \text{（不包含偏置项）}$$

最终，在完成梯度的计算后，我们可以沿着梯度相反的方向更新权重了：

$$\boldsymbol{W}^{(l)} := \boldsymbol{W}^{(l)} - \eta \Delta^{(l)}$$

综上，我们通过下图对反向传播进行总结：



$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W) = a_j^{(l)} \delta_i^{(l+1)}$
（梯度计算）
（输出层误差项）
$\delta^{(3)} = a^{(3)} - y$
1
1
输出（x）
输出（$\hat{y}$）
目标y
$\delta^{(2)} = (W^{(2)})^{\mathrm{T}} \delta^{(3)} * \frac{\partial g(z^{(2)})}{\partial z^{(2)}}$
（隐层误差项）

12.4 建立对反向传播的直观认识

反向传播算法在30年前被重新提出并得到推广，它目前仍是高效训练人工神经网络最为广泛的算法之一。本节我们将对此算法进行更为直观的总结，并对它的工作方式有个全面的了解。

从本质上讲，反向传播仅仅是一种高效地计算复杂代价函数导数的方法。我们的目标是使用这些导数学习权重系数，以对多层人工神经网络进行参数化。神经网络参数化过程中面临的一个挑战就是：我们通常要处理高维特征空间中的数量众多的权重系数。与前面章节中介绍的代价函数不同，神经网络代价函数的误差平面并非是平滑凸曲面。在此高维代价函数的平面上有很多凹陷（局部最优值）和凸起，我们必须克服此问题以找到代价函数的全局最优解。

读者可能会想起微积分基础课程中的链式法则。链式法则是对复杂嵌套函数求导的一种方法，例如，f（g（x））=y可分解为基本的组成部分：

$$\frac{\partial y}{\partial x}=\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

基于代数，已找到一系列方法来高效解决此类问题，这些方法也称作自动微分。如果读者有兴趣了解更多关于自动微分在机器学习中的应用，建议参考如下资源：A.G.Baydin and B.A.Pearlmutter.Automatic Differentiation of Algorithms for Machine Learning.arXiv preprint arXiv:1404.7456,2014，此文献可通过访问arXiv网站（http://arxiv.org/

pdf/1404.7456.pdf）免费获取。

自动微分包含两种模式，分别为：正向积分模式和反向积分模式，反向传播仅为反向模式自动微分的一个特例。关键在于：由于需要将每一层的大型矩阵（雅可比矩阵，Jacobians）与一向量相乘以获得输出，因此在正向模式中使用链式法则的计算成本是非常巨大的。反向模式的诀窍在于：我们从右边开始，将一个矩阵与一个向量相乘，得到的结果是另一个向量，再将其与下一矩阵相乘，如此反复。矩阵-向量的计算成本要远小于矩阵-矩阵相乘的成本，这也是反向传播成为神经网络训练中最常用算法的原因之一。

12.5 通过梯度检验调试神经网络

实现人工神经网络是相当复杂的，而手动检查已实现的反向传播算法是否正确向来被视作一个好的方法。在本节中，我们将讨论一个称作梯度检验（gradient checking）的简单过程，它本质上是网络解析梯度与数值梯度之间的比较。梯度检验并不只限定于前馈神经网络，它也适用于其他基于梯度优化的神经网络架构。即使读者要实现基于梯度优化的普通算法，如线性回归、逻辑斯谛回归，以及支持向量机等，使用梯度检验来验证一下梯度的计算是否正确也是一个不错的主意。

在上一节中，我们将代价函数定义为J（W），其中W为人工神经网络的权重系数矩阵。请注意：粗略来说，J（W）是包含一个隐层的多层感知器中W（1）和W（2）的堆叠。我们将W（1）定义为h×[m+1]维的矩阵，通过它将输入层连接到隐层，其中h为隐层元素的数量，而m为特征（输入单元）的数量。而连接隐层与输出层的W（2）矩阵维度为t×h，其中t为输出单元的数量。进而计算代价函数对于权重$w_{i,j}^{(l)}$的导数：

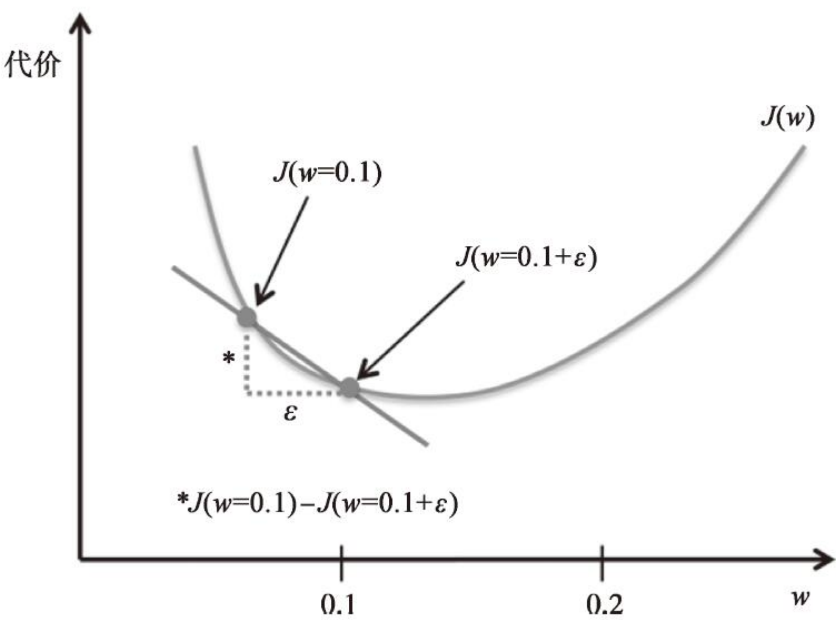
$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(\boldsymbol{W})$$

请记住，我们沿着梯度相反的方向进行权重的更新。在梯度检验中，我们将此分析结果与一个数值近似梯度进行比较：

$$\frac{\partial}{\partial w_{i,j}^{(l)}} J(\boldsymbol{W}) \approx \frac{J(w_{i,j}^{(l)}+\varepsilon)-J(w_{i,j}^{(l)})}{\varepsilon}$$

这里，ε通常是一个极小的数值，如1e-5（请注意这里1e-5仅是

0.00001的更为简洁的记法）。直观上看，我们将此有限的近似差异看作是连接代价函数上两个权重w和w+ε（二者均为标量）之间直线的斜率。为了简单起见，我们忽略了上下标：



得到一个更为准确的梯度近似值的方法是：计算两点间代价函数之差与对应两点间横坐标距离之商：

$$\frac{J(w_{i,j}^{(l)}+\varepsilon)-J(w_{i,j}^{(l)}-\varepsilon)}{2\varepsilon}$$

通常情况下，L2向量范数可以通过数值梯度J'_n与解析梯度J'_a之差来进行计算。在实践中，我们通常将经过计算的梯度矩阵展开为一个向量，这样可以更方便地计算误差（梯度向量之间的差异）：

$$误差=\|J'_n-J'_a\|_2$$

这里有一个问题：随着尺度的变动，误差值并非不变（当权重向量范数很小的时候，即便很小的误差也会感觉很明显）。由此，对误差进行归一化处理：

$$相对误差=\frac{\|J_n'-J_a'\|_2}{\|J_n'\|_2+\|J_a'\|_2}$$

现在，我们希望数值梯度与解析梯度之间的相对误差尽可能小。在实现梯度检验之前，我们还需仔细考虑一个问题：以多大的可接受误差作为通过梯度验证的阈值？通过梯度检验的相对误差阈值与网络架构的复杂性息息相关。一般来说，在正确实现反向传播算法的前提下，我们添加的隐层数量越多，数值梯度与解析梯度间的差异就越大。由于我们在本章中实现的神经网络比较简单，因此可以对阈值做一个相对严格的限制，规则如下：

·相对误差小于等于1e-7意味着一切正常！

·相对误差小于等于1e-4意味着条件可能存在问题，需要检查。

·相对误差大于1e-4意味着代码中可能存在某些错误。

我们已经建立好了基本规则，现在开始实现梯度检验。为了做到这一点，我们可以简单使用前面实现的NeuralNetMLP类，并在类中增加如下方法：

```
def _gradient_checking(self, X, y_enc, w1,
                       w2, epsilon, grad1, grad2):
    """ Apply gradient checking (for debugging only)

    Returns
    ---------
    relative_error : float
      Relative error between the numerically
```

```
    approximated gradients and the backpropagated gradients.

    """
    num_grad1 = np.zeros(np.shape(w1))
    epsilon_ary1 = np.zeros(np.shape(w1))
    for i in range(w1.shape[0]):
        for j in range(w1.shape[1]):
            epsilon_ary1[i, j] = epsilon
            a1, z2, a2, z3, a3 = self._feedforward(
                                            X,
                                            w1 - epsilon_ary1,
                                            w2)
            cost1 = self._get_cost(y_enc,
                                   a3,
                                   w1-epsilon_ary1,
                                   w2)
            a1, z2, a2, z3, a3 = self._feedforward(
                                          X,
                                          w1 + epsilon_ary1,
                                          w2)
            cost2 = self._get_cost(y_enc,
                                   a3,
                                   w1 + epsilon_ary1,
                                   w2)
            num_grad1[i, j] = (cost2 - cost1) / (2 * epsilon)
            epsilon_ary1[i, j] = 0

    num_grad2 = np.zeros(np.shape(w2))
    epsilon_ary2 = np.zeros(np.shape(w2))
    for i in range(w2.shape[0]):
        for j in range(w2.shape[1]):
            epsilon_ary2[i, j] = epsilon
            a1, z2, a2, z3, a3 = self._feedforward(
                                             X,
                                             w1,
                                             w2 - epsilon_ary2)
            cost1 = self._get_cost(y_enc,
                                   a3,
                                   w1,
                                   w2 - epsilon_ary2)
            a1, z2, a2, z3, a3 = self._feedforward(
                                           X,
                                           w1,
                                           w2 + epsilon_ary2)
            cost2 = self._get_cost(y_enc,
                                   a3,
                                   w1,
                                   w2 + epsilon_ary2)
            num_grad2[i, j] = (cost2 - cost1) / (2 * epsilon)
            epsilon_ary2[i, j] = 0

    num_grad = np.hstack((num_grad1.flatten(),
                          num_grad2.flatten()))
    grad = np.hstack((grad1.flatten(), grad2.flatten()))
    norm1 = np.linalg.norm(num_grad - grad)
    norm2 = np.linalg.norm(num_grad)
```

```
    norm3 = np.linalg.norm(grad)
    relative_error = norm1 / (norm2 + norm3)
    return relative_error
```

_gradient_checking方法中的代码看起来相当简单。我的个人建议就是尽量保持代码的简单易懂。我们的目的是仔细检查梯度计算，因此要保证在梯度检验中，不会因为代码高效却复杂，就带来额外错误。接下来，仅需对fit方法做稍许修改。出于清晰易读的考虑，下面的代码中省略了fit函数开始部分的代码，只需将注释##start gradient checking与##end gradient checking之间列出的代码加入到原来实现的fit方法中：

```
class MLPGradientCheck(object):
    [...]
    def fit(self, X, y, print_progress=False):
        [...]
                # compute gradient via backpropagation
                grad1, grad2 = self._get_gradient(
                                       a1=a1,
                                       a2=a2,
                                       a3=a3,
                                       z2=z2,
                                       y_enc=y_enc[:, idx],
                                       w1=self.w1,
                                       w2=self.w2)

                ## start gradient checking
                grad_diff = self._gradient_checking(
                                     X=X[idx],
                                     y_enc=y_enc[:, idx],
                                     w1=self.w1,
                                     w2=self.w2,
                                     epsilon=1e-5,
                                     grad1=grad1,
                                     grad2=grad2)
                if grad_diff <= 1e-7:
                    print('Ok: %s' % grad_diff)
                elif grad_diff <= 1e-4:
                    print('Warning: %s' % grad_diff)
                else:
                    print('PROBLEM: %s' % grad_diff)

                ## end gradient checking

                # update weights; [alpha * delta_w_prev]
                # for momentum learning
                delta_w1 = self.eta * grad1
                delta_w2 = self.eta * grad2
                self.w1 -= (delta_w1 +\
                           (self.alpha * delta_w1_prev))
                self.w2 -= (delta_w2 +\
                           (self.alpha * delta_w2_prev))
                delta_w1_prev = delta_w1
                delta_w2_prev = delta_w2

        return self
```

假定将更改后的多层感知器类命名为MLPGradientCheck，我们现在来初始化一个新的包含10个隐层的多层感知器。同时，我们禁用了正则化、自适应学习，以及动量学习。此外，通过将minibatches的值设置为1来使用常规梯度下降算法。代码如下：

```
>>> nn_check = MLPGradientCheck(n_output=10,
                                n_features=X_train.shape[1],
                                n_hidden=10,
                                l2=0.0,
                                l1=0.0,
                                epochs=10,
                                eta=0.001,
                                alpha=0.0,
                                decrease_const=0.0,
                                minibatches=1,
                                random_state=1)
```

梯度检验的一个缺点就是，它的计算成本极其昂贵。引入梯度检验会使神经网络的训练过程变得非常缓慢，因此我们只希望在算法调试时才用到它。基于此原因，仅使用少量训练样本进行梯度检验屡见不鲜（在此，我们使用了5个样本）。代码如下：

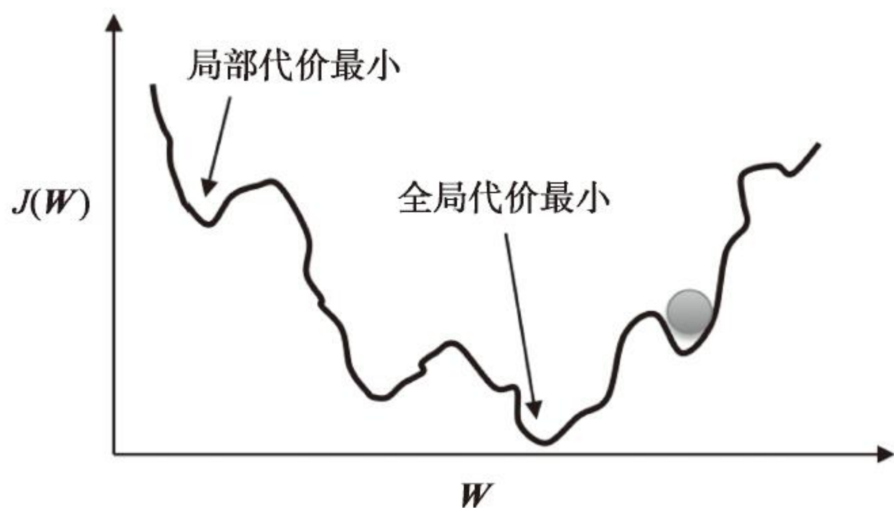
```
>>> nn_check.fit(X_train[:5], y_train[:5], print_progress=False)
Ok: 2.56712936241e-10
Ok: 2.94603251069e-10
Ok: 2.37615620231e-10
Ok: 2.43469423226e-10
Ok: 3.37872073158e-10
Ok: 3.63466384861e-10
Ok: 2.22472120785e-10
Ok: 2.33163708438e-10
Ok: 3.44653686551e-10
Ok: 2.17161707211e-10
```

从代码的输出结果来看，我们的多层感知器完美通过了测试。

12.6 神经网络的收敛性

在训练识别手写数字的神经网络过程中，没有使用传统梯度下降，而是用小批次学习来替代，读者对此可能会感到困惑。请回忆一下在线学习中曾经使用过的随机梯度下降。我们每次仅使用一个训练样本（$k = 1$）更新权重进行。虽然这是一种随机的方法，但与传统梯度下降相比，它通常能够得到精度极高的训练结果，并且收敛速度更快。子批次学习是随机梯度下降的一个特例：从包含n个样本的训练数据集中随机抽取样本数量为k的子集用于训练，其中$1 < k < n$。相较于在线学习，子批次学习的优势在于它能够以向量的方式进行实现，并且能够提高计算效率。然而，仍旧能比传统梯度下降更快地完成权重更新。更直观地说，你可以把小批次学习看作是在一个具有代表性的人口子集上，预测总统选举的投票人数，而不是基于所有的人口进行预测。

此外，我们还额外增加了一些超参，如下降常数以及用于自适应学习速率的参数。原因在于，与简单的算法（如Adaline、逻辑斯谛回归以及支持向量机）相比，神经网络的训练难度更大。在多层神经网络中，通常包含成百上千，甚至能多达10亿个待优化的权重。让人头痛的是，输出函数的曲线并不平滑，而且容易陷入局部最优值，如下图所示：



请注意，上面是一个简化后的示意图，通常情况下神经网络的维度极高，其输出函数曲线的粗糙程度已经超出了人眼识别的范围。在这里，我们展示了在x轴上只包含一个权重的代价函数曲线。这里想传达的信息是，我们不希望算法陷入局部最优解。通过加大学习速率，我们可以轻松地跳出局部最优解。但是，如果学习速率过大，则算法可能会越过全局最优点。由于权重的初始值是随机的，意味着我们要给出一个优化方案解决一个完全错误的问题。通过前面定义的下降常数，我们可以在最初的优化阶段就快速达到目标代价的位置，而自适应学习速率则能帮助我们更好地实现全局最优点。

12.7　其他神经网络架构

本章我们讨论了当前最流行的前馈神经网络中的一种——多层感知器。神经网络是目前机器学习领域中最活跃的研究课题之一，介绍其他神经网络架构超出了本书的范围。如果读者有兴趣了解更多关于深度学习的神经网络和算法，推荐阅读Y.Bengio的论文[1]的介绍和概述部分；Yoshua Bengio由此出版的著作可通过以下链接免费获得：http://www.iro.umontreal.ca/~bengioy/papers/ftml_book.pdf。

神经网络相关的内容足以用一整本书来进行讲解，在此，我们只简单介绍一下另外两种神经网络架构：卷积神经网络（convolutional neural network）和循环神经网络（recurrent neural network）。

[1] Y. Bengio. Learning Deep Architectures for AI. Foundations and Trends in Machine Learning, 2（1）：1-127，2009.

12.7.1　卷积神经网络

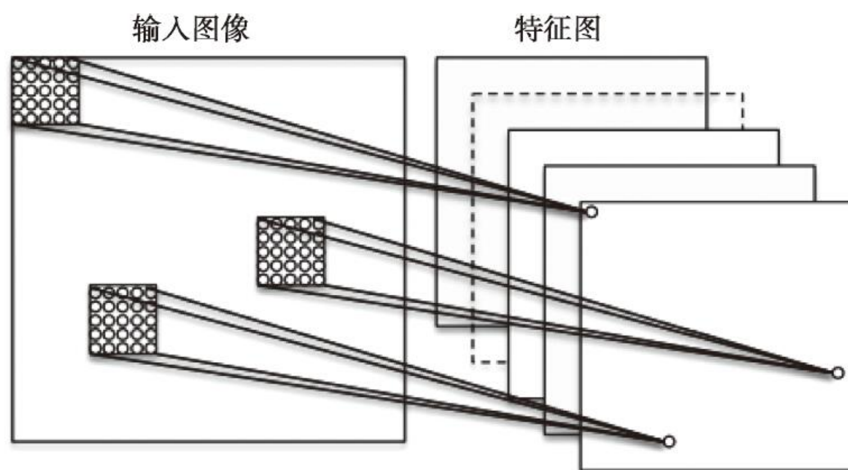
卷积神经网络（Convolutional Neural Network，简写为CNN或ConvNet）在图像识别中的优异表现使其在计算机视觉领域日渐流行，它是当下深度学习领域最流行的神经网络架构之一。卷积神经网络的核心理念在于构建多层特征检测器，以处理输入图片中像素间的空间排列。请注意，卷积神经网络有多个变种。在本节中，我们将讨论此架构的通用理念。如果读者有兴趣了解更多关于卷积神经网络的内容，建议阅读Yann LeCun（http://yann.lecun.com）的相关著作，Yann是卷积神经网络的提出者之一。我特别推荐从下列文献入手学习卷积神经网络：

·Y.LeCun,L.Bottou,Y.Bengio,and P.Haffner.Gradient-based Learning Applied to Document Recognition.Proceedings of the IEEE,86（11）:2278-2324,1998.

·P.Y.Simard,D.Steinkraus,and J.C.Platt.Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis.IEEE,2003,p.958.

回忆一下实现多层感知器时，我们将图像展开为向量，并通过输入层使用全连接将其连接到了隐层——此网络架构没有对空间信息进行编码。在卷积神经网络中，我们使用感受野（receptive field）将输入层连接到特征图（feature map）。可以将感受野理解为层叠的窗口，窗口在可以输入图像的像素上滑动，以此创建特征映射。窗口滑动的幅度以及窗口的大小都是模型的先验超参。对特征图的创建称作卷积（convolution）。连接输

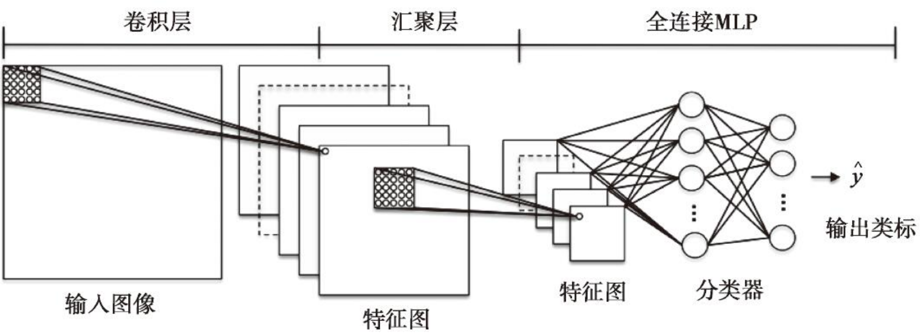
入像素与特征图单元的层称作卷积层，如下图所示：



需要特别注意的是，特征检测器是可以复用的，这意味着将特征转换到像素单元的感受野在下一层中将共享权重。这里关键一点就是：如果特征检测器在图像中的某一位置是有效的，则它在其他位置同样有效。这种方法的优点在于它极大地减少了参数的数量。由于图像中的不同分块可以采用不同的方式来表示，因此卷积神经网络对于识别图片中不同大小、位置的对象效果很好。由于MNIST数据库中的图片已经做过缩放和中心定位，我们无需对此过多担心。

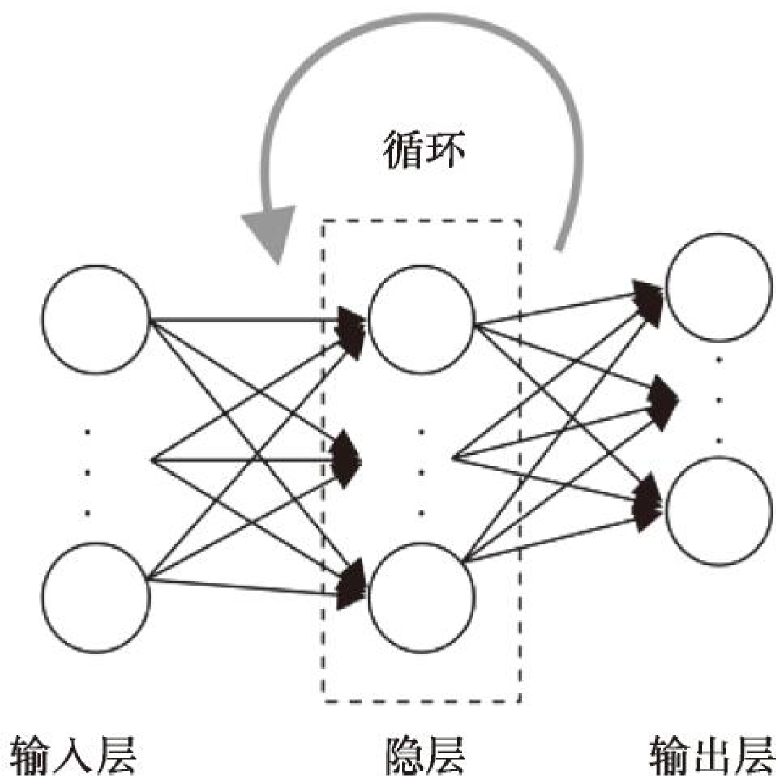
在卷积神经网络中，卷积层之后紧跟的是汇聚层（pooling layer）（有时也称作子采样层（sub-sampling））。在汇聚层中，我们汇总邻近的特征检测器，以减少传入下一层的特征数量。汇聚层可以看作是一种简单的特征抽取，我们可以将临近特征的平均值或最大值抽取出来，并传递给下一层。为了构建深层卷积神经网络，在将神经网络连至多层感知器进行分类之前，我们先在卷积层和汇聚层之间交替地进行多个层的堆叠。如

下图所示：



12.7.2　循环神经网络

循环神经网络（Recurrent Neural Network，RNN）可被理解为包含与时间相关的反馈循环或者反向传播的前馈神经网络。在循环神经网络中，神经元（暂时）释放前，它们只能在有限的时间内处于活动状态。反过来，这些神经元在下一个时间点又会激活其他神经元使之处于活动状态。本质上讲，我们可将循环神经网络看作包含额外时间变量的多层感知器。由于具备这样的时间属性和自身的动态结构，使得网络不光能接受当前的输入值，还可以接受此前的其他输入。



尽管循环神经网络在语音识别、语言翻译，以及手写识别等领域成效显著，不过这种网络架构训练起来却相当困难。这是因为我们不能简单地将错误逐层反向传播，而必须考虑到时间维度，它放大了梯度消失与梯度爆炸的问题。在1997年，Juergen Schmidhuber和他同事提出了所谓的长短时记忆（long short-term memory）单元来解决此问题[1]。

此外，我们还应注意，循环神经网络有许多种不同类型的变体，关于它们的详细讨论已经超出了本书的范围。

[1] S. Hochreiter and J. Schmidhuber. Long Short-term Memory. Neural Computation, 9(8):1735-1780, 1997.

12.8 关于神经网络的实现

我们没有使用开源的Python机器学习库来实现识别手写数字的功能，而是在学习所有相关理论后通过实现一个简单的多层人工神经网络来完成此任务，读者对此可能会感到困惑。其中一个原因是：在撰写本书内容时，scikit-learn还没有加入多层感知器相关的功能。更重要的是，作为机器学习从业者，我们至少应当对自己正在使用的算法有个最基本的了解，为以后更好地运用机器学习算法打好基础。

至此，我们已经了解了前馈神经网络的工作原理，可以进一步去尝试一些更加复杂的基于NumPy的Python库，如Theano（http://deeplearning.net/software/theano/），它可以帮助我们更加高效地构建神经网络。我们将在第13章中对其进行介绍。在过去的几年里，Theano得到了许多机器学习研究者的青睐，由于它能够使用图形处理器（Graphical Processing Unit，GPU）为多维数组的计算进行数学表达式的优化，因此广泛应用于构建深度神经网络。

读者可通过链接http://deeplearning.net/software/theano/tutorial/index.html#tutorial获取大量关于Theano的入门教程。

此外，还有几个与使用Theano训练神经网络相关的库，目前这些库都处于积极开发的状态，读者应多多关注：

·Pylearn2（http://deeplearning.net/software/pylearn2/）

·Lasagne（https://lasagne.readthedocs.org/en/latest/）

·Keras (http://keras.io)

本章小结

在本章，我们学习了多层人工神经网络中最重要的一些概念，神经网络是当前机器学习理论研究领域最热门的话题。在第2章，从单个神经元入手开启了机器学习之旅，而在本章，我们将多个神经元连接为一个功能强大的神经网络，它可以解决诸如手写数字识别等复杂问题。本章还揭秘了流行的反向传播算法，它是众多神经网络模型的基石之一，这些模型可用于构建深度网络。在学习了反向传播算法后，通过对权重更新的实践完成了对神经网络的训练。其中，也做了一些有益的修改，例如加入子批次学习和自适应学习速率等，以更高效地训练神经网络。

第13章　使用Theano并行训练神经网络

在上一章中，通过许多数学概念了解了前馈人工神经网络以及多层感知器在特定任务中是如何工作的。首要一点是，较好地掌握机器学习相关的数学基础理论是至关重要的，它可以保证正确且高效地使用那些功能强大的算法。纵观上一章，我们在机器学习实践环节花费了很多的时间和精力，并且尝试了从零开始实现相关算法。在本章，读者可以稍微轻松一点，我们将开始一个令人兴奋的机器学习之旅——使用一个强大的广受机器学习研究者欢迎的库进行深度网络相关的实验，并对其进行高效训练。在现代的机器学习研究中，计算机通常配有强大的图形处理器（GPU）。如果读者对当前机器学习领域最热门的深度学习感兴趣，本章内容就能满足你的需要。不过，如果读者没有功能强大的独立显卡也无需担心，这只是可选项，并非必需的。

在开始之前，我们先来看一下本章大体将涵盖哪些内容：

·使用Theano编写优化的机器学习代码

·为人工神经网络选择合适的激励函数

·使用Keras深度学习库进行快速便捷的实验

13.1　使用Theano构建、编译并运行表达式

在本节，我们将初步了解强大的Theano库，它能让我们使用Python高效地进行机器学习模型的训练。Theano最初是由Joshua Bengio领导的LISA（Laboratoire d'Informatique des Systèmes Adaptatifs）实验室[1]于2008年开发的。

在深入探究Theano到底是什么，以及它是如何提高机器学习任务处理速度之前，我们先来讨论一下当硬件运行极度耗费运算能力的程序时将会面临的挑战。幸运的是，计算机处理器的性能多年来持续提高，这使得我们可以训练更加强大、复杂的学习系统，从而提高机器学习模型的预测性能。即使是现在最便宜的计算机，其中央处理器也是多核的。在前面章节中，我们看到scikit-learn中的许多函数都可以通过使用多核处理器来提高计算性能。默认情况下，由于全局解释器锁（Global Interpreter Lock，GIL）的缘故，Python代码在执行时只能使用其中一个核。不过，尽管可以使用多线程运行（multiprocessing）库将计算分布到多个核上并行执行，我们也必须意识到，即使最先进的桌面硬件也很少能提供超过8个或者16个核心的计算单元。

回想上一章中，我们实现了一个非常简单的多层感知器模型，虽然它只有一个包含50个节点的隐层，但我们已经需要通过学习来优化大约1000个权重以完成一个非常简单的图像识别任务。MNIST库中的图像是非常小的（只有28×28像素），对于像素密度更高的图像来说，如果我们通过增加额外的隐层对其进行处理，那么可以想象参数数量将会呈现爆炸式

增长。随着图像像素的增加，仅使用一个CUP来对其进行处理很快就会变得不可行。现在面临的问题是我们如何才能更加高效地处理此类问题？一个明显的解决方案就是使用GPU。GPU的性能确实很强大，读者可以将显卡看作是我们电脑中的一个小型计算机集群。另一个优势就是，与最新的CPU相比，GPU价格相对低廉，具体比较信息如下图所示：

性能指标	英特尔®酷睿™i7-5960X Extreme Edition 处理器	英伟达 GeForce® GTX™ 980 Ti GPU
标准频率	3.0 GHz	1.0 GHz
核心数量	8	2816
内存带宽	68 GB/s	336.5 GB/s
浮点数计算	354 GFLOPS	5632 GFLOPS
购买费用	1000 美元	700 美元

以上信息来源于如下网站（2015年8月20日）：

·http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980-ti/specifications

·http://ark.intel.com/products/82930/Intel-Core-i7-5960X-Processor-Extreme-Edition-20M-Cache-up-to-3_50-GHz

一块价格只有CPU 70%的GPU，内核数量却是前者的450倍，而且每秒浮点运算次数也是前者的15倍以上。那么是什么原因阻碍了我们使用GPU来执行机器学习任务呢？我们面临的挑战是：编写执行于GPU的代码不像在解释器中执行Python代码那样方便快捷，我们需要特殊的功能包，如CUDA和OpenCL等。但是，对于实现并运行机器学习算法来说，编写基于CUDA或者OpenCL的代码并不是最好的方案。好消息是，开发出Theano就是用来解决GPU上机器学习的问题。

[1] http://lisa.iro.umontreal.ca.

13.1.1 什么是Theano

Theano究竟是什么呢？一种编程语言？一个编译器？抑或是一个Python库？实际上，Theano兼具以上几种功能。Theano针对于多维数组（张量），能够高效地实现、编译和评估数学表达式。它同时还可以使得代码在GPU运行。不过，其强大性能的真实来源则是利用了GPU中巨大内存带宽以及GPU浮点数运算能力。我们使用Theano可以很容易地共享内存来并行运行代码。Theano开发者在2010年公布的测试报告中指出：在CPU上执行程序时，Theano程序的性能是NumPy的1.8倍，而在GPU上执行相应的代码，Theano速度是NumPy的11倍[1]。请注意，上面是2010年的测试结果，近几年来，无论是Theano还是显卡的性能都有了显著的提高。

那么，Theano和NumPy之间到底有何关系呢？Theano建立在NumPy的基础上并且与其语法相似，这使得熟悉NumPy的用户用起Theano来也得心应手。公正地说，Theano并不像许多人所描述的那样，是“NumPy的兴奋剂”，不过与SymPy（http://www.sympy.org）相类似，SymPy是一个用于符号计算（或者称其为符号代数）的Python包。正如前面章节中所介绍的，我们使用NumPy来描述变量，并组合这些变量，然后再逐行执行相关代码。不过在Theano中，我们首先要写出问题，然后分析问题给出描述。如果代码需要在GPU上运行，Theano可以使用C/C++，或者CUDA/OpenCL生成相应的代码。为了能够生成最优化的代码，我们需要指出待解决问题的领域，可以将其看作是由许多操作构成的树（或者是关于图的符号表达）。请注意，Theano当前仍处于开发活跃阶段，会定期加入新的

功能或更新现有功能。在本章，我们首先探究Theano背后的基本概念，然后进一步学习如何将其用于机器学习实践中。由于Theanno是一个包含许多先进功能的库，我们无法在本书中覆盖其所有内容。不过，如果读者希望了解更多关于Theano的内容，可以访问在线文档（http://deeplearning.net/software/theano/）获取更加详细的信息。

[1] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y.Bengio. Theano: A CPU and GPU Math Compiler in Python. In Proc. 9th Python in Science Conf, pages 1-7,2010.

13.1.2　初探Theano

本节中，我们将迈出学习Theano的第一步。根据系统的设置，通常可以使用pip安装Theano，请在命令行窗口中执行如下命令：

```
pip install Theano
```

对于安装过程中可能出现的问题，建议读者通过如下链接：http://deeplearning.net/software/theano/install.html，了解关于系统和平台方面的具体建议。本章中的所有代码均可在CPU中运行；虽然GPU是可选的，但如果读者想体会Theano的真实性能，建议选配。如果读者拥有支持CUDA或者OpenCL的显卡，请参照相关最新教程http://deeplearning.net/software/theano/tutorial/using_gpu.html#using-gpu进行合理的配置。

张量是Theano的核心元素，Theano使用张量对符号数学表达式进行评估。张量可以看作是标量、向量、矩阵等的泛化。具体而言，标量可以定义为0阶张量，而向量和矩阵则可以分别定义为1阶张量和2阶张量，而在第三维上对矩阵的叠加则为3阶张量。作为热身，我们先通过Theano的tensor模块中的标量来计算一维数据样本点x的净输入z，其中权重为w_1，偏置值为w_0：

$$z=x_1\times w_1+w_0$$

代码如下：

```
>>> import theano
>>> from theano import tensor as T

# initialize
>>> x1 = T.scalar()
>>> w1 = T.scalar()
>>> w0 = T.scalar()
>>> z1 = w1 * x1 + w0

# compile
>>> net_input = theano.function(inputs=[w1, x1, w0],
...                             outputs=z1)

# execute
>>> print('Net input: %.2f' % net_input(2.0, 1.0, 0.5))

 Net input: 2.50
```

这种方法是不是简单直接？如果我们在Theano写代码，只须遵循三个步骤：定义符号（变量对象），编译代码，执行代码。初始化阶段，我们定义了三个符号：x1、w1和w0，用来计算z1。然后对函数net_input进行编译，计算出净输入z1。

不过，当我们开发基于Theano的代码时，有一个需要特别注意的细节：变量的类型（dtype）。无论如何，我们在使用整型或者浮点型数据时，需要考虑选择使用64位还是32位的表示方式，这将对代码性能产生极大的影响。我们将在下一节中对此进行更深入的讨论。

13.1.3　配置Theano

无论是macOS、Linux，还是微软的Windows操作系统，系统和应用程序主要都使用64位内存寻址方式。但是，如果想要使用GPU加速评估数学表达式，我们依然依赖于32位的内存寻址方式。这是目前Theano唯一支持的计算架构。本节中，我们将学习如何合理地配置Theano。如果读者对与Theanno配置相关的更多细节感兴趣，请参考在线文档：http://deeplearning.net/software/theano/library/config.html，当实现机器学习算法时，我们主要使用浮点数。默认情况下，NumPy和Theano都使用双精度浮点格式（float64）。不过，当我们在CPU上使用Theano代码开发原型，并将其放到GPU上执行运算时，将浮点数精度在float64（CPU）和float32（GPU）之间来回转换是非常有用的。例如，在Python交互环境下，我们可以执行下面的代码查看Theano中浮点变量的默认设置：

```
>>> print(theano.config.floatX)
float64
```

如果读者在安装Theano后未对设置做任何修改，浮点数的默认设置应为64位。不过，我们可以使用下列代码将其在当前Python交互下的设置更改为32位：

```
>>> theano.config.floatX = 'float32'
```

请注意，虽然Theano目前在GPU上使用32位浮点类型，但在CPU上64位和32位浮点类型都可以使用。因此，如果读者想要更改全局默认设置，可以在命令行（Bash）终端中修改THEANO_FLAGS变量的值：

```
export THEANO_FLAGS=floatX=float32
```

此外，也可以将此设置写入特定的Python脚本中，并通过以下方式运行脚本：

```
THEANO_FLAGS=floatX=float32 python your_script.py
```

至此，我们已经讨论了如何设置浮点数的默认格式，使得Theano在GPU上发挥最大的性能。接下来，我们讨论代码的执行位置在CPU和GPU之间切换的配置选项。执行如下代码，我们可以检查当前使用的是CPU还是GPU：

```
>>> print(theano.config.device)
cpu
```

我的个人建议是默认使用cpu，这样设计原型和调试代码就更加容易。例如，在CPU上，我们可以在命令行终端中以脚本的方式来运行Theano代码：

```
THEANO_FLAGS=device=cpu,floatX=float64 python your_script.py
```

当我们完成了对程序的编码，并希望尽可能发挥GPU的最大性能来高效运行这些代码时，可以执行如下代码，它无须对原始程序做任何修改：

```
THEANO_FLAGS=device=gpu,floatX=float32 python your_script.py
```

我们可以在个人的主目录下创建一个.theanorc文件来永久保存这些设置。例如，如果想始终使用32位浮点格式和GPU，可以通过下述命令创建一个包含此设置的.theanorc的文件：

```
echo -e "\n[global]\nfloatX=float32\ndevice=gpu\n" >> ~/.theanorc
```

如果读者并未使用macOS或者Linux终端，可以使用自己熟悉的文本

编辑器来手工创建一个.theanorc文件，文件内容如下：

```
[global]
floatX=float32
device=gpu
```

现在我们已经知道了如何根据硬件情况对Theano做适当配置，在下一节，我们将讨论如何使用更加复杂的阵列结构。

13.1.4 使用数组结构

在本节，我们将讨论如何通过Theano中的tensor模块使用数组结构。执行下列代码，我们可以创建一个简单的2×3矩阵，并使用优化过的Theano张量表达式计算数组中各列之和：

```
>>> import numpy as np

# initialize
>>> x = T.fmatrix(name='x')
>>> x_sum = T.sum(x, axis=0)

# compile
>>> calc_sum = theano.function(inputs=[x], outputs=x_sum)

# execute (Python list)
>>> ary = [[1, 2, 3], [1, 2, 3]]
>>> print('Column sum:', calc_sum(ary))
Column sum: [ 2.  4.  6.]

# execute (NumPy array)
>>> ary = np.array([[1, 2, 3], [1, 2, 3]],
...                dtype=theano.config.floatX)
>>> print('Column sum:', calc_sum(ary))
Column sum: [ 2.  4.  6.]
```

如前所述，使用Theano只需遵循3个基本步骤：定义变量、编译以及执行代码。上例显示Theano可以处理Python和NumPy的数据格式：列表（list）和数组（numpy.ndarry）

注意：在创建TensorVariable的fmatrix变量时，我们使用了可选的名称参数（在此为x），这对调试代码或者输出Theano图来说非常有用。例如，若未使用此名称参数，则输出变量名为x的fmatrix时，实际显示结果是其类型TesnsorType：

```
>>> print(x)
<TensorType(float32, matrix)>
```

但如果在初始化TensorVariable时，像示例代码那样指定了名称参数x，则会显示此参数：

```
>>> print(x)
x
```

此时可通过type方法获取TensorType：

```
>>> print(x.type())
<TensorType(float32, matrix)>
```

Theano还拥有一个非常智能的内存管理系统，通过对内存的复用使得系统运行速度更快。具体来说，Theano可以管理散布在多种硬件上的存储空间，如CPU和GPU。通过跟踪内存空间的变化，它可以对缓冲区进行镜像。接下来，我们来看一下shared变量，它允许我们使用大型对象（数组），这些对象可以被多种类型的函数读写，这样我们就可以在编译后对更新这些对象。关于Theano内存处理的更详细介绍超出了本书的范围。读者可通过链接http://deeplearning.net/software/theano/tutorial/aliasing.html获取Theano内存管理的最新信息。

```
# initialize
>>> x = T.fmatrix('x')
>>> w = theano.shared(np.asarray([[0.0, 0.0, 0.0]],
                                  dtype=theano.config.floatX))
>>> z = x.dot(w.T)
>>> update = [[w, w + 1.0]]

# compile
>>> net_input = theano.function(inputs=[x],
...                             updates=update,
...                             outputs=z)

# execute
>>> data = np.array([[1, 2, 3]],
...                 dtype=theano.config.floatX)
>>> for i in range(5):
...     print('z%d:' % i, net_input(data))
z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]
```

正如我们所见，通过Theano共享内存是很容易的。在上一个例子中，我们定义了一个update变量，并通过此变量声明：在for循环每次迭代后，数组w加1以更新。在定义了要更新的对象，以及如何进行更新之后，我们便可以通过update参数将其传递给theano.function进行编译。

使用Theano的另一个巧妙方法就是：在编译之前通过givens变量在图中加入新的值[1]。使用这种方法，借助于共享变量，我们可以降低通过CPU将数据从内存传输到GPU的次数，从而加速算法的学习速度。如果我们在theano.function中使用了inputs参数，数据将会在CPU和GPU之间多次传输，例如，在梯度下降中，我们需对数据进行多次循环遍历（迭代）。如果数据集能够加载到GPU内存中，使用givens它就可以在训练过程中得以保持（例如，通过子批次进行学习）。代码如下：

```
# initialize
>>> data = np.array([[1, 2, 3]],
...                 dtype=theano.config.floatX)
>>> x = T.fmatrix('x')
>>> w = theano.shared(np.asarray([[0.0, 0.0, 0.0]],
...                              dtype=theano.config.floatX))
>>> z = x.dot(w.T)
>>> update = [[w, w + 1.0]]

# compile
>>> net_input = theano.function(inputs=[],
...                             updates=update,
...                             givens={x: data},
...                             outputs=z)

# execute
>>> for i in range(5):
...     print('z:', net_input())
z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]
```

由上述示例代码，我们还可发现：givens值的格式是Python字典，它可以将变量名映射到真实的Python对象上。其中，变量名与我们在fmatrix中定义的名字相同。

[1] Theano会将计算过程编译成一个图模型。——译者注

13.1.5 整理思路——线性回归示例

现在我们已经对Theano有了初步的认识，再来看一个非常实际的例子，并实现一个基于最小二乘法（Ordinary Least Squares，OLS）的回归。关于回归分析的相关内容，请参见第10章。

让我们从创建包含5个训练样本的一维数据集开始：

```
>>> X_train = np.asarray([[0.0], [1.0],
...                       [2.0], [3.0],
...                       [4.0], [5.0],
...                       [6.0], [7.0],
...                       [8.0], [9.0]],
...                     dtype=theano.config.floatX)
>>> y_train = np.asarray([1.0, 1.3,
...                       3.1, 2.0,
...                       5.0, 6.3,
...                       6.6, 7.4,
...                       8.0, 9.0],
...                     dtype=theano.config.floatX)
```

请注意，在构造NumPy数组时，我们将数据格式设定为theano.config.floatX，这使得我们可以在需要时将数据在CPU和GPU之间来回传递。

接下来，我们来实现一个训练函数，以误差平方和作为代价函数，它可以通过学习得到线性回归模型的回归系数。其中，w_0为偏置单元（$x = 0$时y轴上的截距）。代码如下：

```
import theano
from theano import tensor as T
import numpy as np

def train_linreg(X_train, y_train, eta, epochs):

    costs = []
    # Initialize arrays
    eta0 = T.fscalar('eta0')
    y = T.fvector(name='y')
    X = T.fmatrix(name='X')
    w = theano.shared(np.zeros(
                        shape=(X_train.shape[1] + 1),
                        dtype=theano.config.floatX),
                      name='w')

    # calculate cost
    net_input = T.dot(X, w[1:]) + w[0]
    errors = y - net_input
    cost = T.sum(T.pow(errors, 2))

    # perform gradient update
    gradient = T.grad(cost, wrt=w)
    update = [(w, w - eta0 * gradient)]

    # compile model
    train = theano.function(inputs=[eta0],
                            outputs=cost,
                            updates=update,
                            givens={X: X_train,
                                    y: y_train,})

    for _ in range(epochs):
        costs.append(train(eta))

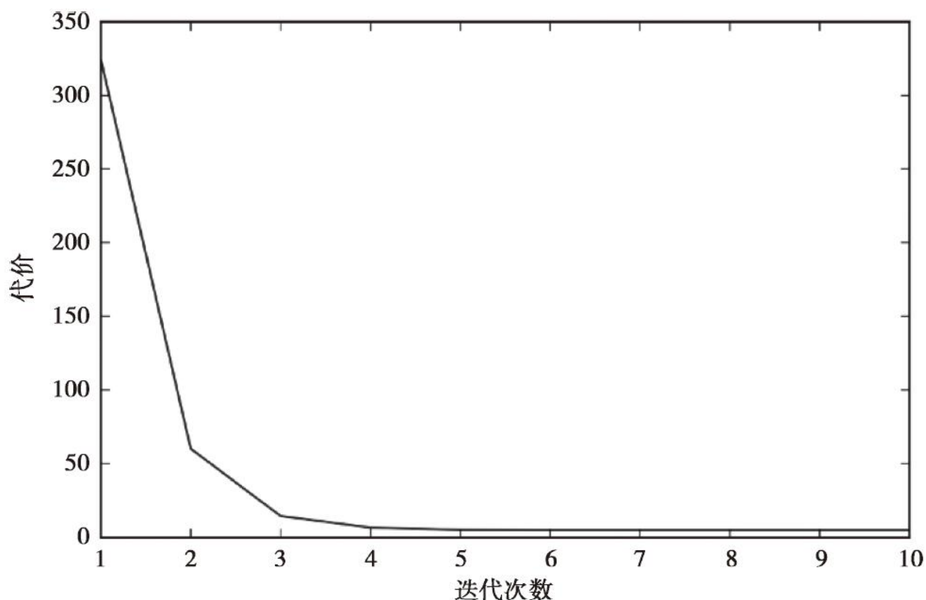
    return costs, w
```

上述示例代码中的grad函数是Theano的一个非常实用的功能。我们通过wrt传递参数，此函数可以自动计算表达式相对于参数的导数。

在实现了训练函数后，我们训练线性回归模型，并通过查看误差平方和代价函数的值来检验模型是否收敛：

```
>>> import matplotlib.pyplot as plt
>>> costs, w = train_linreg(X_train, y_train, eta=0.001, epochs=10)
>>> plt.plot(range(1, len(costs)+1), costs)
>>> plt.tight_layout()
>>> plt.xlabel('Epoch')
>>> plt.ylabel('Cost')
>>> plt.show()
```

如下图所示，学习算法在经过5次迭代后就已经收敛：



到目前为止一切进展顺利，通过查看代价函数，我们似乎已经在此特定数据集上构建了一个可用的回归模型。现在，我们编译一个新的函数，用来对输入的特征进行预测：

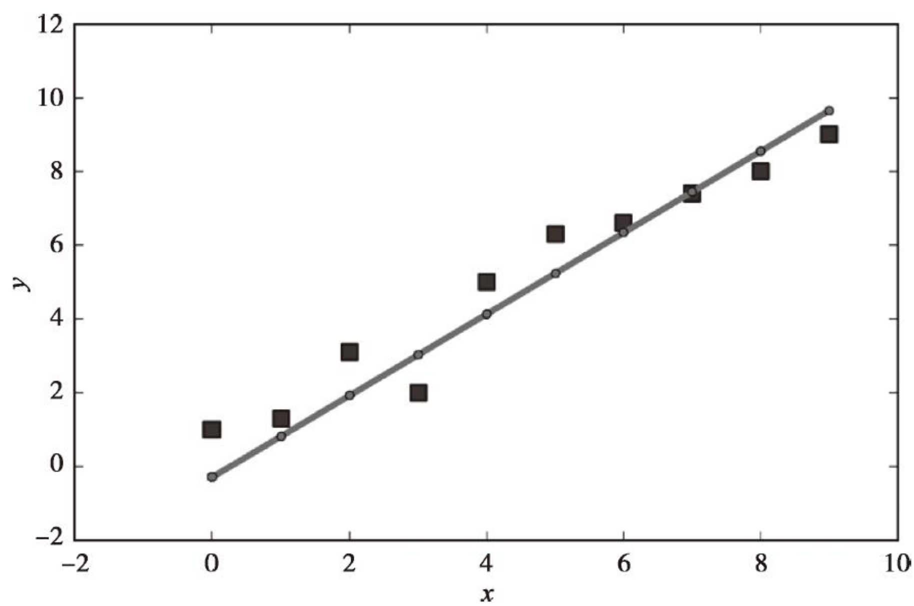
```
def predict_linreg(X, w):
    Xt = T.matrix(name='X')
    net_input = T.dot(Xt, w[1:]) + w[0]
    predict = theano.function(inputs=[Xt],
                              givens={w: w},
                              outputs=net_input)
    return predict(X)
```

实现预测函数也相当简单，只需遵循Theano的三个步骤：定义、编译和执行。接下来，我们绘制出线性回归在训练数据上的拟合情况：

```
>>> plt.scatter(X_train,
...             y_train,
...             marker='s',
...             s=50)
>>> plt.plot(range(X_train.shape[0]),
...          predict_linreg(X_train, w),
...          color='gray',
...          marker='o',
...          markersize=4,
...          linewidth=3)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.show()
```

由结果图像可以看出，模型恰当地拟合了数据点。

构建简单的回归模型是熟悉Theano API的一个很好的方式。不过，我们的最终目标是发挥出Theano的优势，也就是实现高效的人工神经网络。现在我们已经拥有了实现第12章中多层感知器所需的工具：Theano。不过实现起来依然让人感到枯燥，对吗？因此，我们来学习一下我个人最喜欢的一个深度学习库，它构建在Theano之上，可以尽可能简单地实现神经网络相关的实验。不过，在介绍Keras库之前，我们先在下一节中讨论一下如何选择神经网络中的激励函数。



12
10
8
6
4
2
0
−2
y
−2
0
2
4
6
8
10
x

13.2 为前馈神经网络选择激励函数

简单起见，目前为止我们只讨论了多层前馈神经网络中用到的sigmoid激励函数，它应用到了第12章介绍的多层感知器的隐层和输出层中。我们把此激励函数称为sigmoid函数，因为这是学术文献中的常用叫法——更为准确的定义应该是逻辑斯谛函数（logistic function），或者负对数似然函数（negative log-likehood function）。在本节的后续内容中，读者将学到更多的sigmoid系列激励函数，它们常用于实现多层神经网络。

从技术上讲，任何一个函数只要是可微的，我们就可以将其用作多层神经网络的激励函数。在Adaline中，我们甚至可以使用线性激励函数（请见第2章）。不过，实际应用中，在隐层和输出层使用线性激励函数并不能发挥多大的作用，因此我们希望将非线性函数引入到典型的神经网络中，以解决复杂问题。毕竟线性函数的组合得到的还是线性函数。

上一章中使用的逻辑斯谛激励函数（logistic activation function）可能是对人类大脑神经元概念最为近似的模拟：我们可以将其看作神经元是否被激活的概率。不过，当输入为极大的负值时，逻辑斯谛激励函数也会出问题，这种情况下它的输出会趋近于0。如果sigmoid激励函数的输出趋近于0，那么将会降低神经网络的学习速度，并且在训练过程中更容易陷入局部最小值。这也是人们喜欢在隐层使用双曲正切（hyperbolic tangent）函数作为激励函数的原因。在讨论双曲正切函数之前，我们先对逻辑斯谛函数做一个简单的概述，并且分析一下到底是什么特性使得它在

多类别分类任务中起到了如此重要的作用。

13.2.1 逻辑斯谛函数概述

我们前面提到的逻辑斯谛函数，通常称作sigmoid函数，它实际上是sigmoid函数的一个特例。请回顾一下第3章中逻辑斯谛回归一节，在二类别分类任务中，我们使用逻辑斯谛函数对样本x属于正类别（类别1）的概率进行建模：

$$\phi_{logistic}(z) = \frac{1}{1+e^{-z}}$$

其中，标量z被定义为净输入：

$$z = w_0 x_0 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}$$

其中，w_0是偏置单元（当$x_0 = 1$时在y轴上的截距）。再给出一个更为具体的例子：下列代码演示的是一个适用于二维数据点x的模型，其中w为模型的权重系数向量：

```
>>> X = np.array([[1, 1.4, 1.5]])
>>> w = np.array([0.0, 0.2, 0.4])

>>> def net_input(X, w):
...     z = X.dot(w)
...     return z

>>> def logistic(z):
...     return 1.0 / (1.0 + np.exp(-z))

>>> def logistic_activation(X, w):
...     z = net_input(X, w)
...     return logistic(z)

>>> print('P(y=1|x) = %.3f'
...       % logistic_activation(X, w)[0])
P(y=1|x) = 0.707
```

如果使用给定的特征值和权重系数来计算净输入，并用结果去激励逻辑斯谛神经元，我们将得到返回值0.707，意思就是给定的样本x属于正类的概率为70.7%。在第12章，我们使用独热编码技术计算包含多个逻辑斯谛激励单元的输出层的值。不过，正如下列示例代码所示，包含多个逻辑斯谛激励单元的输出层无法提供有意义、可解释的概率值：

```
# W : array, shape = [n_output_units, n_hidden_units+1]
#          Weight matrix for hidden layer -> output layer.
# note that first column (A[:][0] = 1) are the bias units
>>> W = np.array([[1.1, 1.2, 1.3, 0.5],
...               [0.1, 0.2, 0.4, 0.1],
...               [0.2, 0.5, 2.1, 1.9]])

# A : array, shape = [n_hidden+1, n_samples]
#          Activation of hidden layer.
# note that first element (A[0][0] = 1) is the bias unit
>>> A = np.array([[1.0],
...               [0.1],
...               [0.3],
...               [0.7]])

# Z : array, shape = [n_output_units, n_samples]
#          Net input of the output layer.
>>> Z = W.dot(A)
>>> y_probas = logistic(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
 [[ 0.87653295]
 [ 0.57688526]
 [ 0.90114393]]
```

正如输出结果所示，特定样本属于第一个类别的概率大概为88%，此样本属于第二个类别的概率约为58%，而它属于第三个类别的概率为90%。这明显让我们产生了疑惑：各类别的概率之和应该为100%。不过，实际上，比例的值在这里并非大问题，我们只是用模型来预测类标，而不用给出样本属于某个类别的概率。

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('predicted class label: %d' % y_class[0])
predicted class label: 2
```

不过，在某些情况下，在多类别分类中给出所属类别概率是非常有用的。在下一节，我们将介绍更一般化的逻辑斯谛函数——softmax函数，它可以在多类别分类任务中给出样本所属类别的概率。

13.2.2　通过softmax函数评估多类别分类任务中的类别概率

softmax函数是更加一般化的逻辑斯谛函数，在多类别分类（多类别逻辑斯谛回归）中，它使得我们能够计算有意义的类别隶属概率。在softmax中，分母是经归一化处理的所有M个线性函数之和，而分子为净输入z，二者的比值即为特定样本属于第i个类别的概率：

$$P(y=i|z)=\phi_{softmax}(z)=\frac{e_i^z}{\sum_{m=1}^{M} e_m^z}$$

我们使用Python代码来完成一个softmax实例：

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))

>>> def softmax_activation(X, w):
...     z = net_input(X, w)
...     return sigmoid(z)

>>> y_probas = softmax(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
 [[ 0.40386493]
 [ 0.07756222]
 [ 0.51857284]]
>>> y_probas.sum()
1.0
```

正如我们所预期的那样，各类别隶属概率之和为1。值得一提的是，样本属于第二个类别的概率接近于0，这是由于z和max（z）之间存在较大差距。不过总体预测结果与逻辑斯谛函数的结果一致。直观上看，将softmax函数看作是归一化的逻辑斯谛函数，有助于在多类别分类中对类

别隶属做出有意义的预测。

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('predicted class label:
...        %d' % y_class[0])
predicted class label: 2
```

13.2.3 通过双曲正切函数增大输出范围

另一个在人工神经网络隐层中经常用到的sigmoid函数是双曲正切（hyperbolic tangent，tanh）函数，它可以看作是经过缩放的逻辑斯谛函数。

$$\phi_{\tanh}(z)=2\times\phi_{logistic}(2\times z)-1=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

$$\phi_{logistic}(z)=\frac{1}{1+e^{-z}}$$

$$\text{logistic}(2\times z)\times 2-1$$

与逻辑斯谛函数相比，双曲正切函数的优势在于它输出值的范围更广，介于开区间（-1，1），这可以加速反向传播算法的收敛[1]。而逻辑斯谛函数返回值的开区间为（0，1）。为了对逻辑斯谛函数和双曲正切函数做个直观的比较，我们绘制出两个函数在同一区间中的图像：

```
>>> import matplotlib.pyplot as plt

>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)

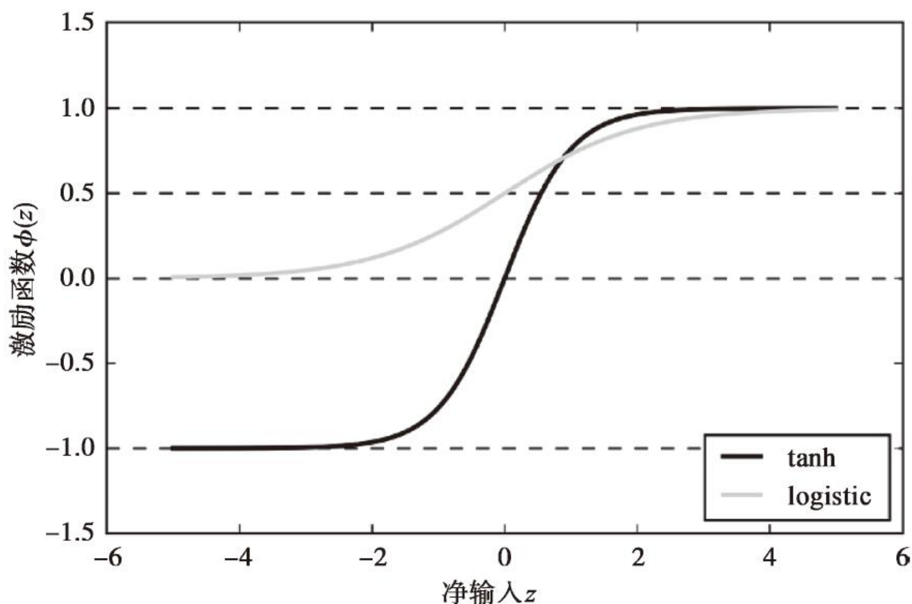
>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)

>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('net input $z$')
>>> plt.ylabel('activation $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle='--')
>>> plt.axhline(0.5, color='black', linestyle='--')
>>> plt.axhline(0, color='black', linestyle='--')
>>> plt.axhline(-1, color='black', linestyle='--')

>>> plt.plot(z, tanh_act,
...          linewidth=2,
...          color='black',
...          label='tanh')
>>> plt.plot(z, log_act,
...          linewidth=2,
...          color='lightgreen',
...          label='logistic')

>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

从图中可以看出，两个S型曲线的形状非常类似；不过，tanh函数的输出范围是逻辑斯谛函数的2倍：



请注意，出于演示的需要，我们在本章中实现了logistic和tanh函数，在实际应用中，我们可以直接使用NumPy中的tanh函数以获得同样的结果：

```
>>>  tanh_act = np.tanh(z)
```

此外，在SciPy的special模块中，已经实现了logistic函数：

```
>>> from scipy.special import expit
>>> log_act = expit(z)
```

我们已经了解了人工神经网络中常用的激励函数，最后，我们总结一下本书中出现过的多种不同的激励函数来结束本小节的内容。

激励函数类型	公　式	示　例	一 维 图 像
单位阶跃函数（heaviside）	$\phi(z)=\begin{cases}0, & z<0\\ 0.5, & z=0\\ 1, & z>0\end{cases}$	感知器模型	
符号函数（signum）	$\phi(z)=\begin{cases}-1, & z<0\\ 0, & z=0\\ 1, & z>0\end{cases}$	感知器模型	
线性函数	$\phi(z)=z$	Adaline，线性回归	
分段函数	$\phi(z)=\begin{cases}1, & z\geqslant\frac{1}{2}\\ z+\frac{1}{2}, & -\frac{1}{2}<z<\frac{1}{2}\\ 0, & z\leqslant-\frac{1}{2}\end{cases}$	支持向量机	
逻辑斯谛（sigmoid）	$\phi(z)=\frac{1}{1+e^{-z}}$	逻辑斯谛回归、多层神经网络	
双曲正切	$\phi(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	多层神经网络	

[1] C. M. Bishop. Neural networks for pattern recognition. Oxford university press, 1995, pp. 500-501.

13.3　使用Keras提高训练神经网络的效率

在本节，我们将讨论一下Keras，它是一个最近开发的用于促进神经网络训练的库。Keras的开发始于2015年年初，时至今日，它已经发展成为构建在Theano之上的最流行、应用最广泛的库，通过Keras，我们可以使用GPU来加速神经网络的训练。Keras的一个突出特性在于它提供了一个非常直观的API，这让我们通过简单的几行代码实现神经网络成为了可能。安装好Theano后，读者可以在命令行终端窗口中通过下面的命令从PyPI中安装Keras：

```
pip install Keras
```

关于Kears的更多信息，请访问其官方网站：http://keras.io。

为了解如何通过Keras进行神经网络的训练，我们来实现一个多层感知器，用于识别前面章节中用到的MNIST数据集中的手写数字。MNIST数据集分为四个部分，可通过链接http://yann.lecun.com/exdb/mnist/进行下载：

·train-images-idx3-ubyte.gz：训练集图片（9912422字节）

·train-labels-idx1-ubyte.gz：训练集图片对应的类标（28881字节）

·t10k-images-idx3-ubyte.gz：测试集图片（1648877字节）

·t10k-labels-idx1-ubyte.gz：测试集图片对应的类标（4542字节）

解压下载的压缩包，将得到的文件都放置在当前工作目录下的mnist

目录中，这样我们就可以使用下列函数加载训练和测试数据集：

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte'
                                % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte'
                               % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                             dtype=np.uint8).reshape(len(labels), 784)
    return images, labels
X_train, y_train = load_mnist('mnist', kind='train')
print('Rows: %d, columns: %d' % (X_train.shape[0], X_train.shape[1]))
Rows: 60000, columns: 784
X_test, y_test = load_mnist('mnist', kind='t10k')
print('Rows: %d, columns: %d' % (X_test.shape[0], X_test.shape[1]))
Rows: 10000, columns: 784
```

在后面的几页中，我们将使用代码示例逐步讲解Keras的使用过程，读者可以在Python解释器中直接输入并执行这些代码。不过，如果读者想要在GPU上训练神经网络，可以将这些代码输入到一个脚本中，或者从Packt出版社的官网[1]下载相关代码。为了在GPU上运行Python脚本，请在mnist_keras_mlp.py文件所在目录执行如下命令：

```
THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python mnist_
keras_mlp.py
```

继续训练数据前的准备工作，现在我们将MNIST图像的数组转换为32

位浮点数格式：

```
>>> import theano
>>> theano.config.floatX = 'float32'

>>> X_train = X_train.astype(theano.config.floatX)
>>> X_test = X_test.astype(theano.config.floatX)
```

接下来，我们将类标（数字0～9）转换为独热码格式。幸运的是，Keras提供了一个便捷的工具来完成格式的转换：

```
>>> from keras.utils import np_utils
>>> print('First 3 labels: ', y_train[:3])
First 3 labels:  [5 0 4]
>>> y_train_ohe = np_utils.to_categorical(y_train)
>>> print('\nFirst 3 labels (one-hot):\n', y_train_ohe[:3])
First 3 labels (one-hot):
 [[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]]
```

现在，我们来到了最有趣的环节：实现一个神经网络。这里使用了与第12章中相同的神经网络结构。不过，我们在隐层和输出层分别使用了双曲正切函数和softmax函数作为激励函数，以替代原来的逻辑斯谛函数同时还额外增加了一个隐层。正如下列代码所示，Keras使得神经网络的实现变得非常简单：

```
>>> from keras.models import Sequential
>>> from keras.layers.core import Dense
>>> from keras.optimizers import SGD

>>> np.random.seed(1)

>>> model = Sequential()
>>> model.add(Dense(input_dim=X_train.shape[1],
...                 output_dim=50,
...                 init='uniform',
...                 activation='tanh'))

>>> model.add(Dense(input_dim=50,
...                 output_dim=50,
...                 init='uniform',
...                 activation='tanh'))

>>> model.add(Dense(input_dim=50,
...                 output_dim=y_train_ohe.shape[1],
...                 init='uniform',
...                 activation='softmax'))

>>> sgd = SGD(lr=0.001, decay=1e-7, momentum=.9)
>>> model.compile(loss='categorical_crossentropy', optimizer=sgd)
```

我们首先使用Sequential类初始化了一个模型来实现前馈神经网络。接下来，我们可以根据需要加入任意数量的层。不过，由于加入的第一个层是输入层，我们必须保证input_dim属性的值与训练集中特征的数量（列）相匹配（在本例中为768）。同时，我们还须保证上一层的输出单元数量（output_dim）与下一层的输入单元数量（input_dim）相匹配。在前面的例子中，我们加入了两个隐层，每个隐层都包含50个隐藏单元及一个偏置单元。请注意，与第12章中将多层感知器的偏置单元默认设置为1（常见但不一定是最好的设置）不同，Keras将全连接网络的偏置单元初始化设置为0。

最后，输出层中单元数量应与类标类别的数量相同（有多少类型的类标就有多少个输出单元）——也就是使用独热码表示的类标数组中列的数量。在编译模型之前，我们还须定义一个优化器。在上面的例子中，我们

选择通过随机梯度下降进行优化，该方式在前面章节已相当熟悉。此外，与第12章中讨论的方法类似，我们可以通过设置训练过程中权重衰减常数与动量学习的值来调整迭代过程中学习的速率。最后，我们将代价（损失）函数设置为categorical_crossentropy。这个（二进制）交叉熵是与逻辑斯谛回归中代价函数相关的一个技术术语。此交叉熵是通过softmax激励函数对多类别分类的泛化。在完成模型的编译后，我们就可以通过调用fit方法来对其进行训练。在此，我们使用了子批次随机梯度，每个子批次中包含300个训练样本。我们通过50次迭代完成多层感知器的训练，通过将verboses的值设置为1，我们便可看到通过代价函数进行优化的训练进度。validation_split参数使用起来也非常方便，它会在每次迭代中预留训练样本（本例中是6000个）的10%，并在迭代后使用这些样本进行验证，这样我们就可以检查模型在训练过程中是否过拟合。

```
>>> model.fit(X_train,
...           y_train_ohe,
...           nb_epoch=50,
...           batch_size=300,
...           verbose=1,
...           validation_split=0.1,
...           show_accuracy=True)
Train on 54000 samples, validate on 6000 samples
Epoch 0
54000/54000 [==============================] - 1s - loss: 2.2290 -
acc: 0.3592 - val_loss: 2.1094 - val_acc: 0.5342
Epoch 1
54000/54000 [==============================] - 1s - loss: 1.8850 -
acc: 0.5279 - val_loss: 1.6098 - val_acc: 0.5617
Epoch 2
54000/54000 [==============================] - 1s - loss: 1.3903 -
acc: 0.5884 - val_loss: 1.1666 - val_acc: 0.6707
Epoch 3
54000/54000 [==============================] - 1s - loss: 1.0592 -
acc: 0.6936 - val_loss: 0.8961 - val_acc: 0.7615
[…]
Epoch 49
54000/54000 [==============================] - 1s - loss: 0.1907 -
acc: 0.9432 - val_loss: 0.1749 - val_acc: 0.9482
```

在训练过程中显示代价函数的值也是非常有用的，这样我们就可以迅速发现训练过程中代价是否呈下降趋势，否则便可提前终止算法并对超参的值进行调优。

为了完成类标的预测，我们可以使用predict_classes方法直接以整数的形式返回类标：

```
>>> y_train_pred = model.predict_classes(X_train, verbose=0)
>>> print('First 3 predictions: ', y_train_pred[:3])
>>> First 3 predictions:  [5 0 4]
```

最后，我们看一下模型在训练和测试集上的准确率：

```
>>> train_acc = np.sum(
...        y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Training accuracy: %.2f%%' % (train_acc * 100))

Training accuracy: 94.51%

>>> y_test_pred = model.predict_classes(X_test, verbose=0)
>>> test_acc = np.sum(y_test == y_test_pred,
...                       axis=0) / X_test.shape[0]
print('Test accuracy: %.2f%%' % (test_acc * 100))
Test accuracy: 94.39%
```

请注意，这里我们只是实现了一个不带参数调优的简单神经网络。如果读者对Keras感兴趣，可通过进一步调节学习速率、动量、权重衰减以及隐层元素的数量等参数对神经网络调优。

Keras是实现神经网络并进而对其进行实验的一个优秀的库，但还有许多其他封装了Theano的库也值得一提。特别是Pylearn2（http://deeplearning.net/software/pylearn2/），它由位于蒙特利尔的LISA实验室所开发。如果读者喜欢简约一点的扩展库，那么Lasagne（https://github.com/Lasagne/Lasagne）也许会引起你的注意，它基于Theano开

发，可通过此框架更好地控制Theano。

[1] http://www.packtpub.com

本章小结

希望读者能够喜欢这令人兴奋的机器学习之旅的最后一章。纵观全书，基本涵盖了机器学习领域中全部的内容，而读者也应该有能力将这些技术应用到解决现实世界中的问题。

我们通过对几种不同类型机器学习任务的概述开启了机器学习之旅：监督学习、强化学习，以及无监督学习。我们讨论了用于分类的几种不同的学习算法，并在第2章中介绍了简单的单层神经网络。然后，我们在第3章中讨论了更加高级的分类算法，在第4和第5章中，我们学习了机器学习中至关重要的环节：数据预处理及数据压缩。请大家牢记：即使是最先进的算法也要受制于训练数据所蕴含的信息。在第6章中，我们学习了关于预测模型构建与评估的最佳实践，这也是机器学习应用中的另一重要环节。如果单个学习算法的性能无法达到我们的预期，通过集成多个算法进行预测也许会有所帮助，我们在第7章中对此进行了讨论。在现代社会中，来自互联网中社交媒体平台的文本文档可能是最有趣的数据形式了，我们在第8章中使用机器学习对此类数据进行了分析。不过，机器学习技术并不仅仅只能应用于离线数据分析，在第9章中，我们学习了如何将机器学习模型嵌入到Web应用中使之能够得到更为广泛的应用。大多数情况下，我们的重点是分类算法，这也许是机器学习领域最为流行的应用。不过，不止如此，我们在第10章中介绍了几种用于预测连续输出值的回归算法。机器学习中另一个令人兴奋的领域就是聚类分析，它可以帮助我们在训练数据没有正确引导结果的情况下发现数据中蕴含的结构，我们在第11章中对此做了介绍。

在本书的最后两个章节中，我们学习了整个机器学习领域最美丽也最令人激动的算法：人工神经网络。虽然深度学习已经超出了本书的范围，但我们希望至少能激起读者对当前这一发展迅速的领域的兴趣。如果读者考虑以机器学习研究作为自己的事业，或者仅仅是追赶这一领域的最新进展，建议阅读此领域顶级专家的著作，例如：Geoff Hinton（http://www.cs.toronto.edu/~hinton/），Andrew Ng（http://www.andrewng.org），Yann LeCun（http://yann.lecun.com），Juergen Schmidhuber（http://people.idsia.ch/~juergen/），以及Yoshua Bengio（http://www.iro.umontreal.ca/~bengioy）等。同时，还请加入scikit-learn、Theano以及Keras的邮件列表以参与到关于这些库及机器学习的有趣讨论中。期待在那里与读者相遇！如果对本书有任何疑问或者需要某些建议与提示，也欢迎读者与作者联系！

希望这本包含了机器学习各方面内容介绍的著作能够让读者觉得学有所值，也希望读者能够真正掌握许多新且有用的技巧来推动职业生涯的发展，同时将这些知识运用到现实世界中去解决实际问题。